Service Mesh & Kafka

Part 3/4 : Distributed Computing, Sidecar Design Pattern & Security

Araf Karsh Hamid, Co-Founder / CTO, MetaMagic Global Inc., NJ, USA

Part 1 : Microservices Architecture

Part 2 : Event Storming and Saga

Agenda

1

Micro Services Introduction

1. Micro Services Characteristics
2. Micro Services System Design Model
3. Monolithic Vs. Micro Services Architecture
4. SOA Vs. Micro Services Architecture
5. App Scalability Based on Micro Services
6. Design Patterns

2

Service Mesh

1. Eight fallacies of Distributed Computing
2. Service Mesh
3. Sidecar Design Pattern
4. Service Mesh – Sidecar Design Pattern
5. Service Mesh – Per Host Design Pattern
6. Service Mesh Software Features
7. Service Mesh – Traffic Control
8. Service Mesh Open Source Infrastructures

3

Apache Kafka

1. Kafka Features
2. Kafka Topic & Durability
3. Kafka Data Structure
4. Kafka Operations
5. Kafka Performance
6. Kafka Case Study

Micro Services Characteristics

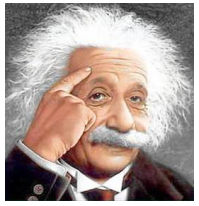
By James Lewis and Martin Fowler

Components via Services

Organized around **Business Capabilities**

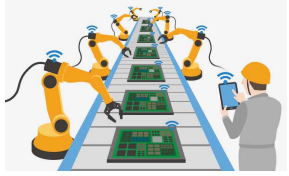
Products NOT Projects

Smart Endpoints & Dumb Pipes



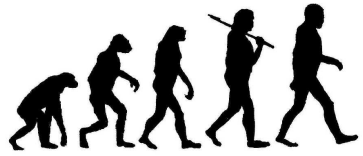
Decentralized Governance & Data Management

Infrastructure **Automation**



Design for **Failure**

Evolutionary Design



The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.
Alan Kay, 1998 email to the Squeak-dev list

Modularity ... is to a technological economy what the division of labor is to a manufacturing one.
W. Brian Arthur,
author of *e Nature of Technology*

We can scale our operation independently, maintain unparalleled system availability, and introduce new services quickly without the need for massive reconfiguration. —
Werner Vogels, CTO, Amazon Web Services

Micro Services System Design Model

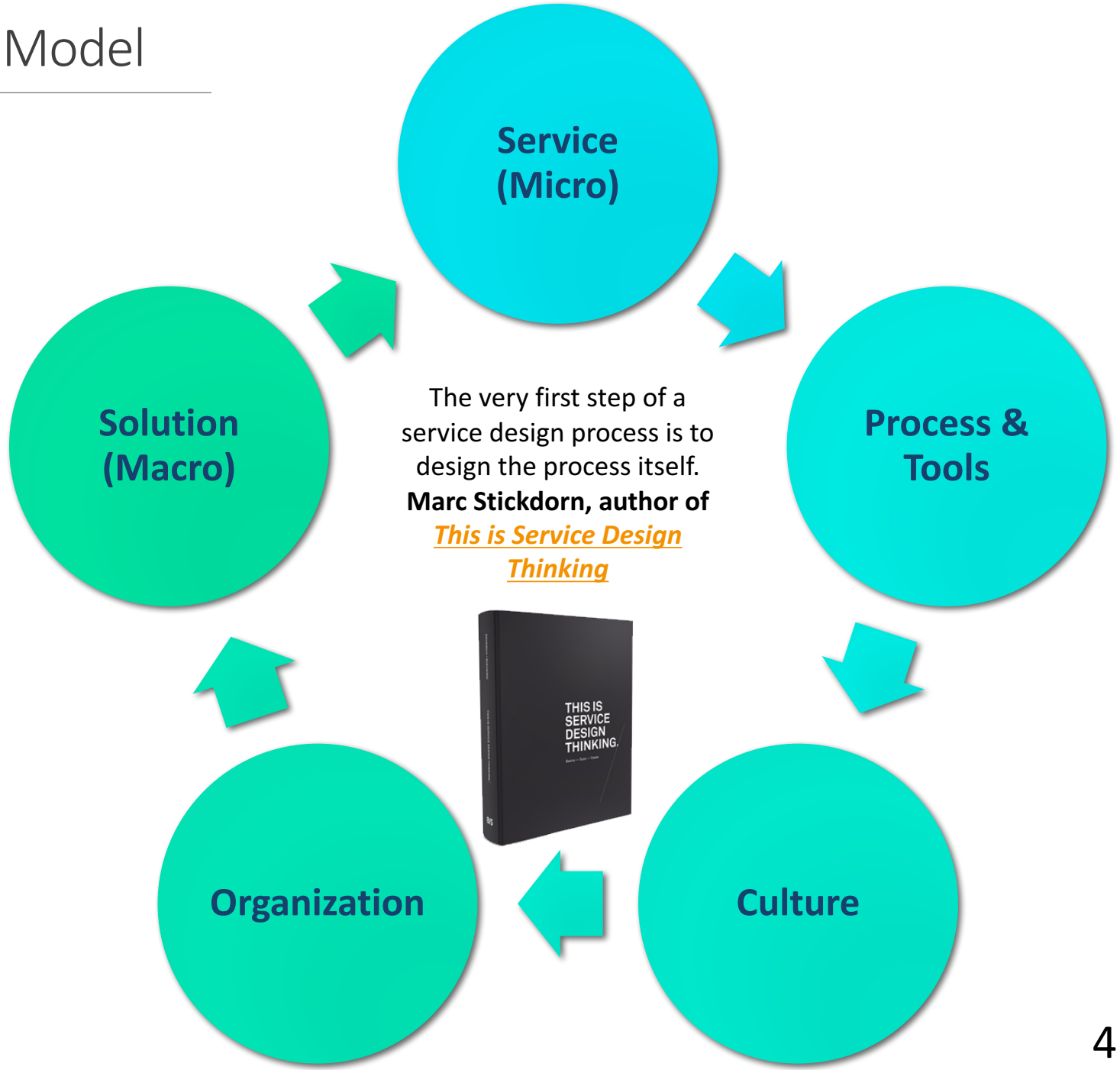
Service: Focuses on a specific Business Capability

Process & Tools: Development, Code Deployment, Maintenance and Product Management

Culture: A Shared set of values, beliefs by everyone. Ubiquitous Language in DDD is an important aspect of Culture.

Organization: Structure, Direction of Authority, Granularity, Composition of Teams.

Solution: Coordinate all inputs and outputs of multiple services. Macro level view of the system allows the designer to focus more on desirable system behavior.



Monolithic vs. Micro Services Example

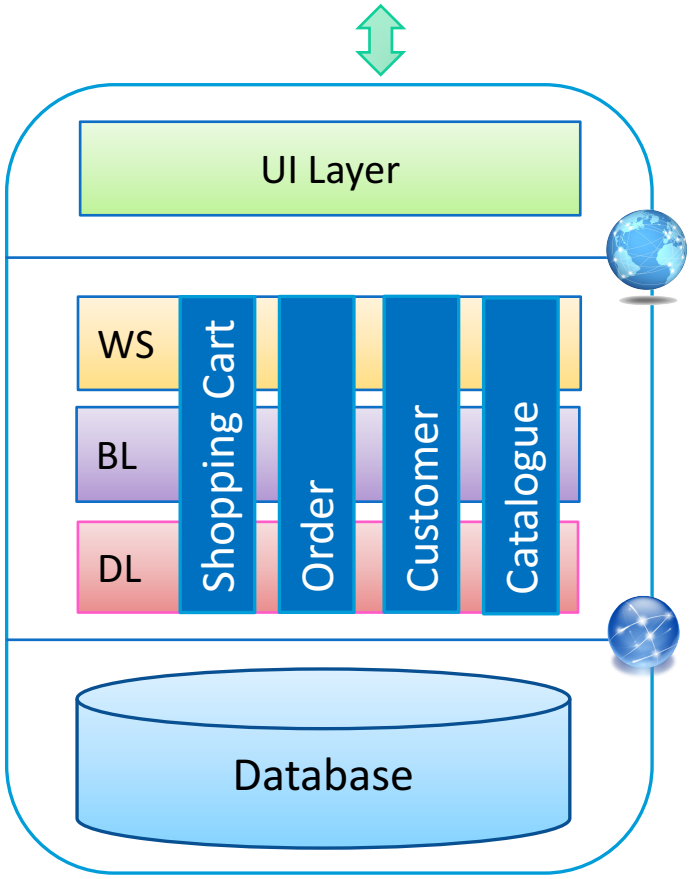




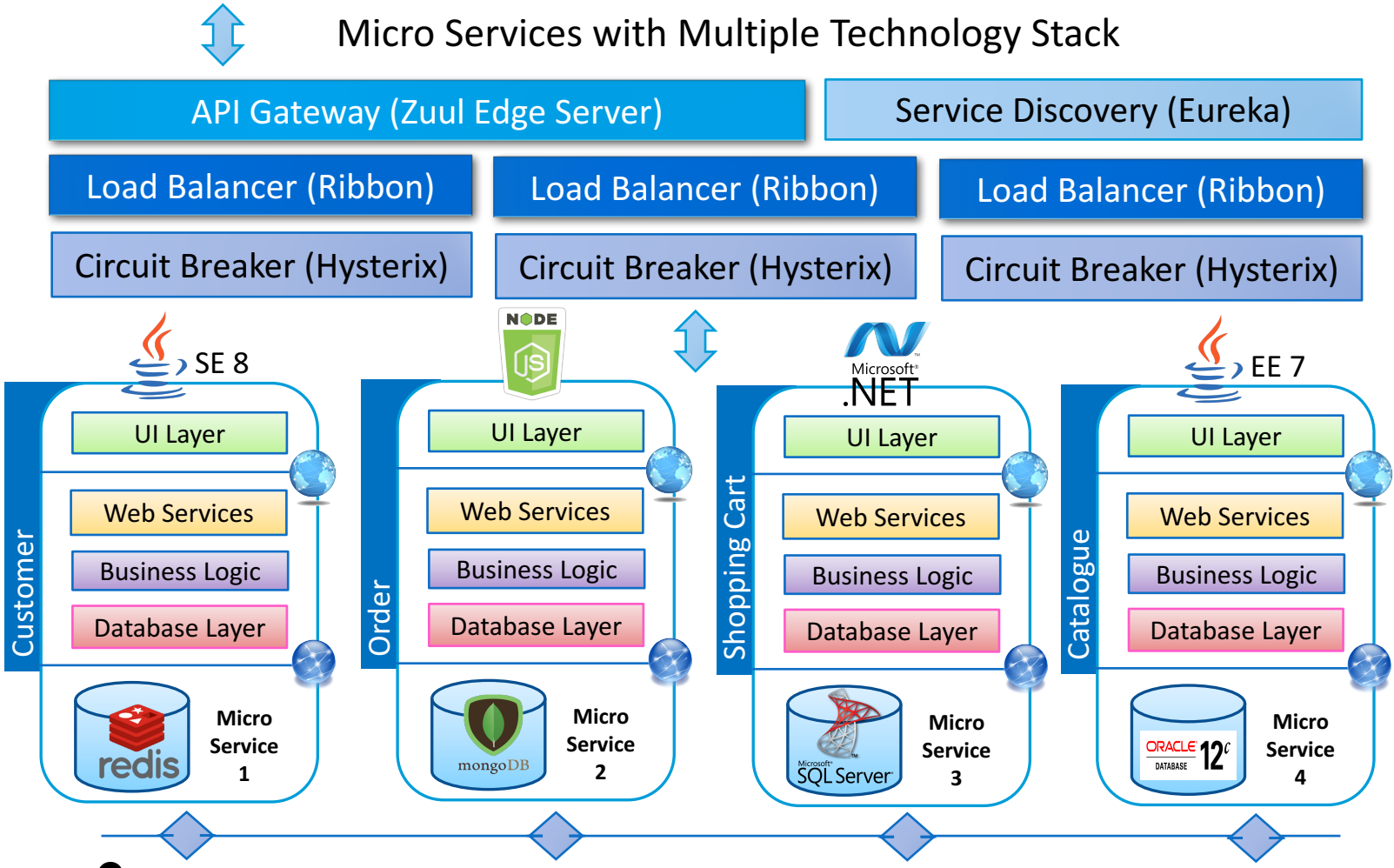
Existing aPaaS vendors creates Monolithic Apps.

This 3 tier model is obsolete now.

Source: Gartner Market Guide for Application Platforms Nov 23, 2016



Traditional Monolithic App using Single Technology Stack



SOA vs. Micro Services Example

Smart Pipes
Lot of Business logic resides in the Pipe

UI Layer

Messaging
REST / SOAP
HTTP
MOM
JMS
ODBC / JDBC
Translation

Enterprise Service Bus

Shopping Cart
Order
Customer
Catalogue

Web Services
XML
WSDL
Addressing
Security
Registry
Management

Producers

Database

Shared Database

Traditional Monolithic App with SOA

Micro Services with Multiple Technology Stack

API Gateway

Service Discovery

Load Balancer | Circuit Breaker

Load Balancer | Circuit Breaker

Load Balancer | Circuit Breaker

SE 8

NODE

Microsoft .NET

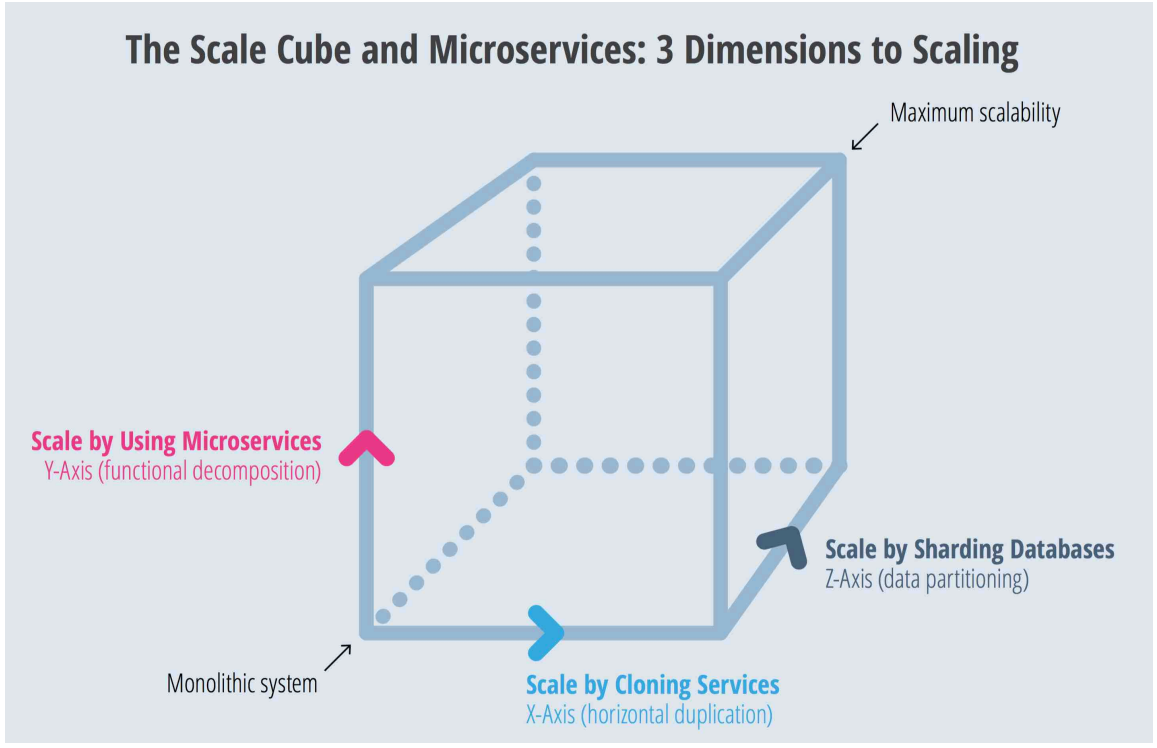
Customer
UI Layer
Web Services
Business Logic
Database Layer
redis
Micro Service 1

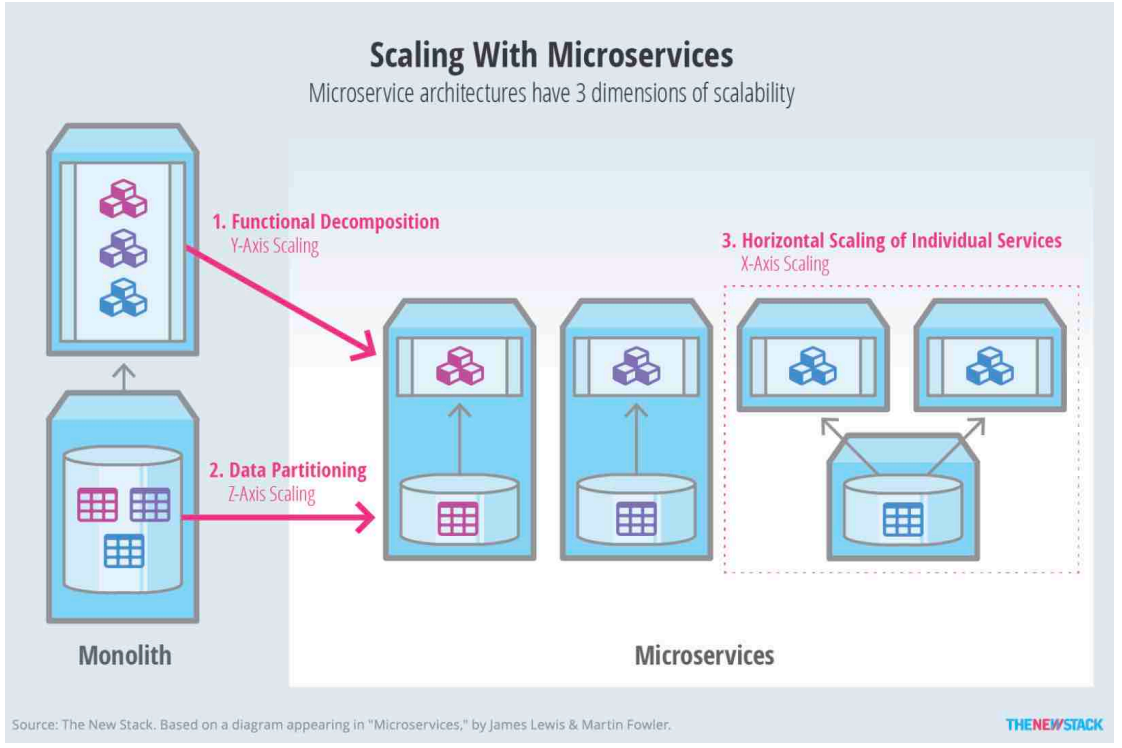
Order
UI Layer
Web Services
Business Logic
Database Layer
mongoDB
Micro Service 2

Shopping Cart
UI Layer
Web Services
Business Logic
Database Layer
Microsoft SQL Server
Micro Service 3

kafka
Event Stream

Scale Cube and Micro Services





1. Y Axis Scaling – Functional Decomposition : Business Function as a Service
2. Z Axis Scaling – Database Partitioning : Avoid locks by Database Sharding
3. X Axis Scaling – Cloning of Individual Services for Specific Service Scalability

Design Patterns

Single Component Pattern

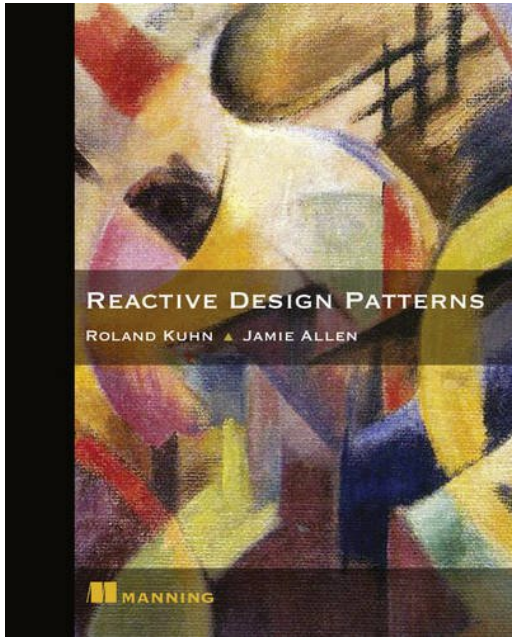
A Component shall do ONLY one thing, But do it in FULL.

Single Responsibility Principle By DeMarco : Structured Analysis & System Specification (Yourdon, New York, 1979)

Saga Pattern

Divide long-lived distributed transactions into quick local ones with compensating actions for recovery.

Pet Helland: Life Beyond Distributed Transactions CIDR 2007

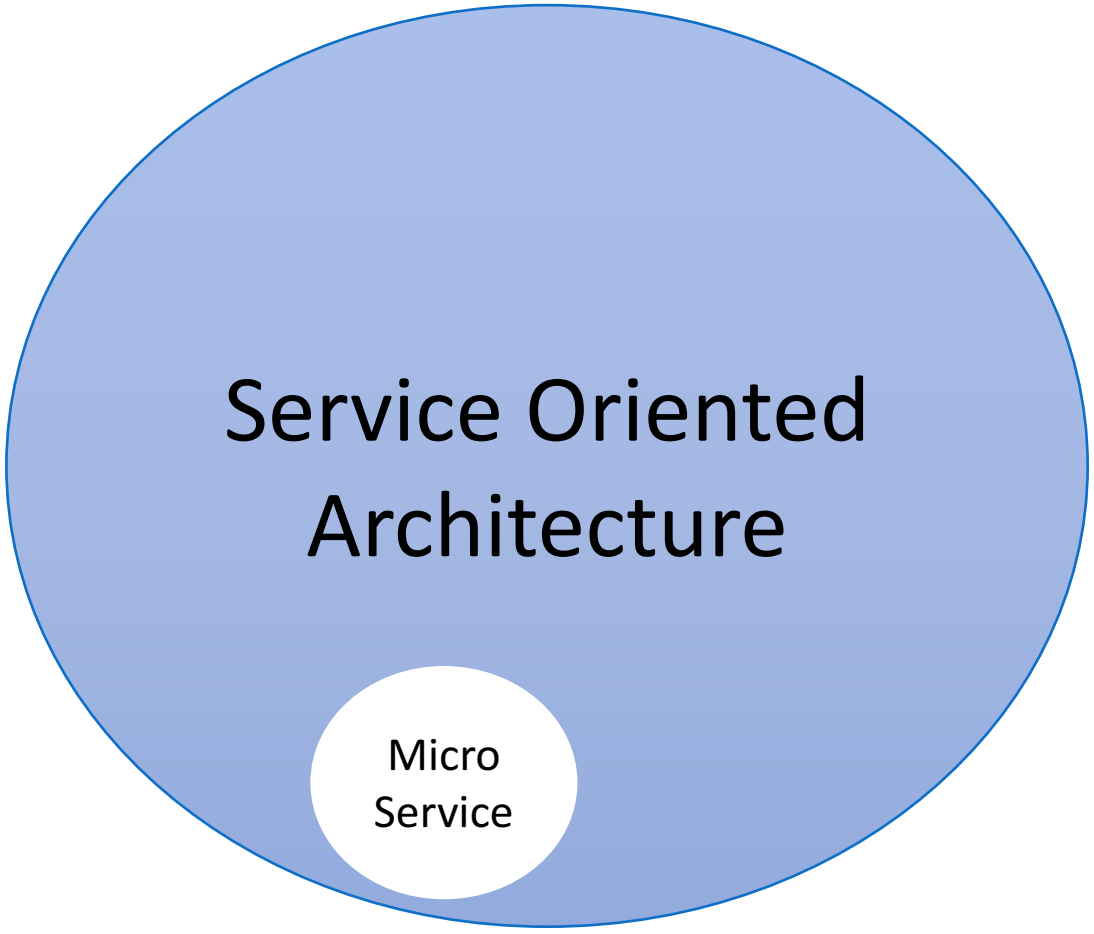


Let-It-Crash Pattern

Prefer a FULL component restart to complex internal failure handling.

Candea & Fox: Crash-Only Software (USENIX HotOS IX, 2003)
Popularized by Netflix Chaos Monkey. Erlang Philosophy

Summary – Micro Services Intro



Microservices Benefits

1. Robust
2. Scalable
3. Testable (Local)
4. Easy to Change and Replace
5. Easy to Deploy
6. Technology Agnostic

Martin Fowler – Micro Services Architecture
https://martinfowler.com/articles/microservices.html
Dzone – SOA vs Micro Services : https://dzone.com/articles/microservices-vs-soa-2

2

Service Mesh

Microservices Communication Infrastructure

1. Eight Fallacies of Distributed Computing
2. Service Mesh
3. Sidecar Design Pattern
4. Service Mesh – Sidecar Design Pattern
5. Service Mesh – Per Host Design Pattern
6. Service Mesh Software Features
7. Service Mesh – Traffic Control
8. Service Mesh Open Source Infrastructures

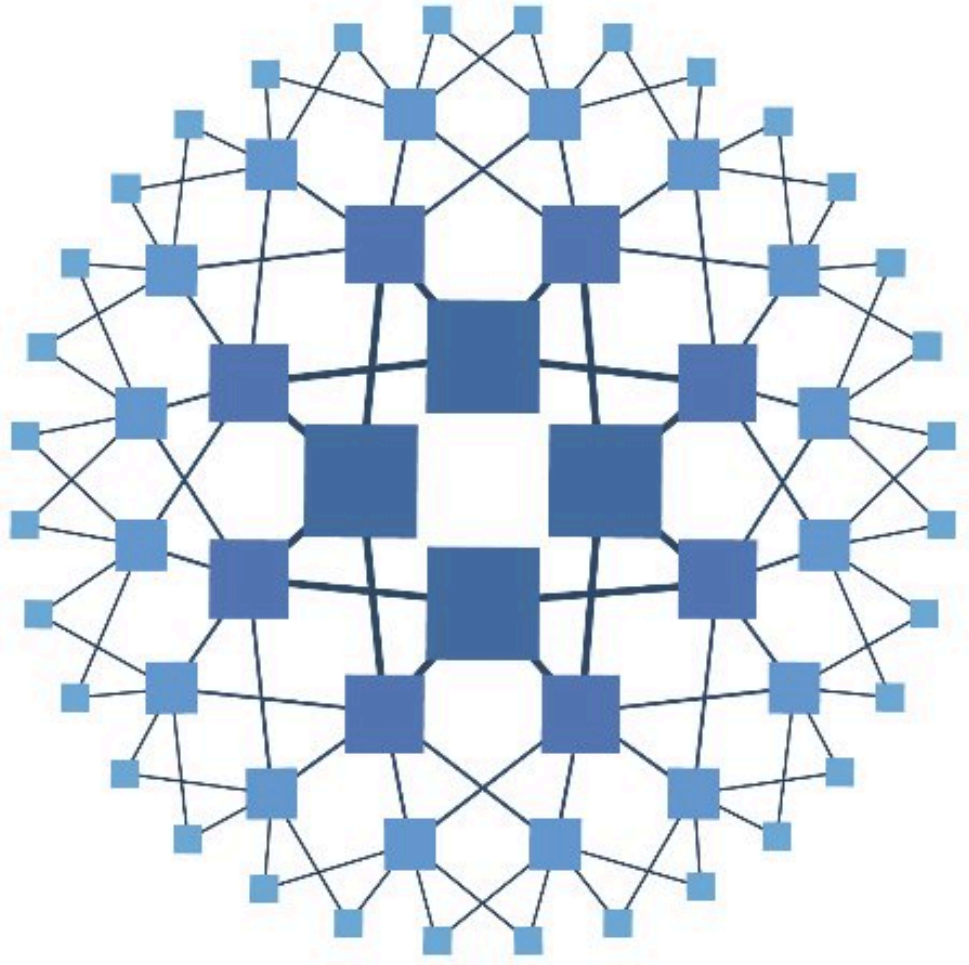
8 Fallacies of Distributed Computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. Transport cost is zero.
5. The network is secure.
6. Topology doesn't change.
7. There is one administrator.
8. The network is homogeneous.

Service Mesh

A service mesh is a dedicated infrastructure layer for making service-to-service communication

1. Safe
2. Reliable
3. Loosely Coupled



Sidecar Design Pattern

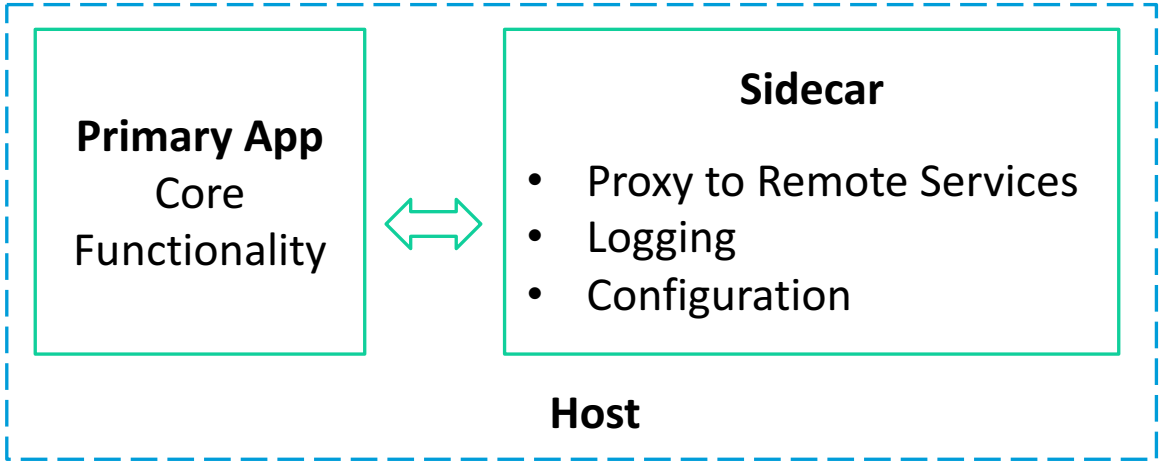
This pattern is named *Sidecar* because it resembles a sidecar attached to a motorcycle

- The sidecar is attached to a parent application and provides supporting features for the application.
- The sidecar also shares the same lifecycle as the parent application, being created and retired alongside the parent.

Context and Problem

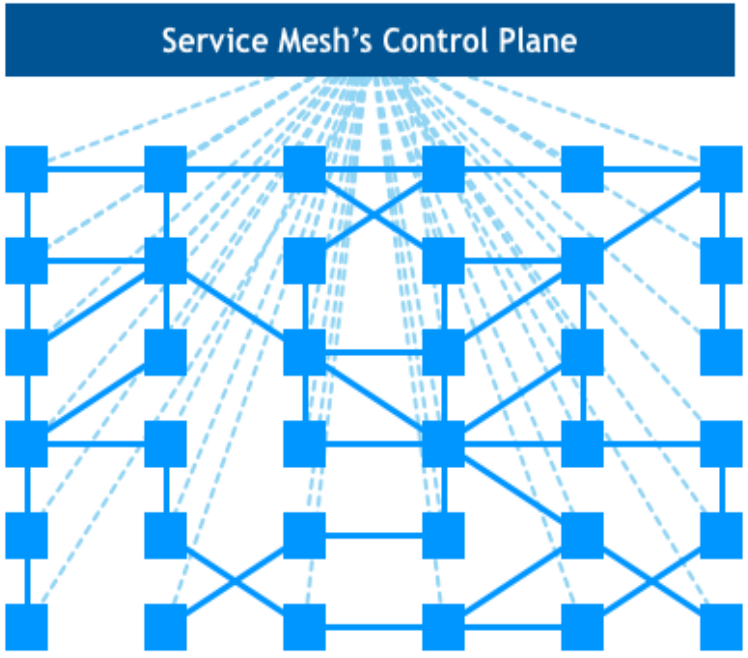
Applications and services often require related functionality, such as monitoring, logging, configuration, and networking services. These peripheral tasks can be implemented as separate components or services.

Solution



Source: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar

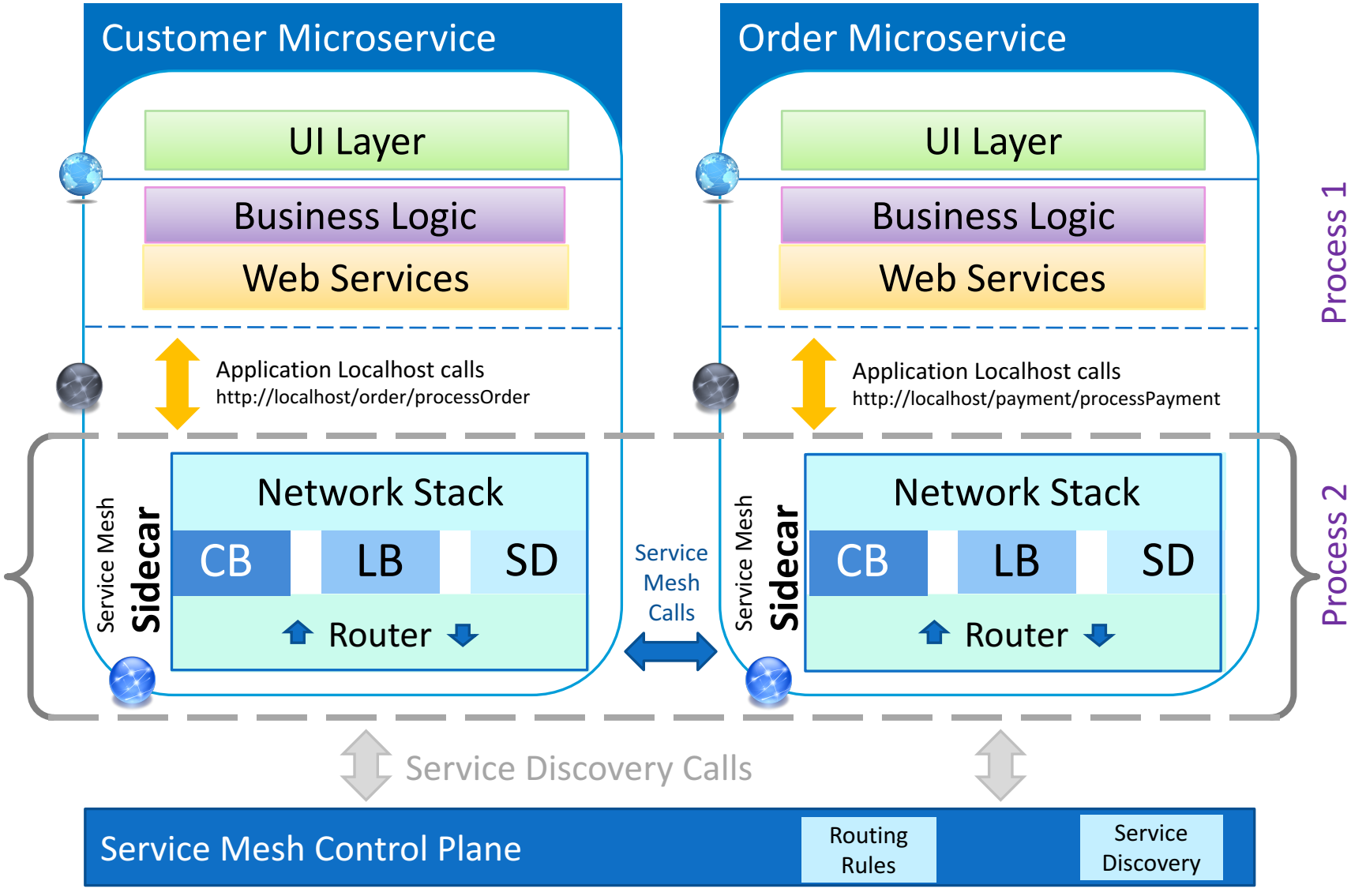
Service Mesh – Sidecar Design Pattern



Microservice

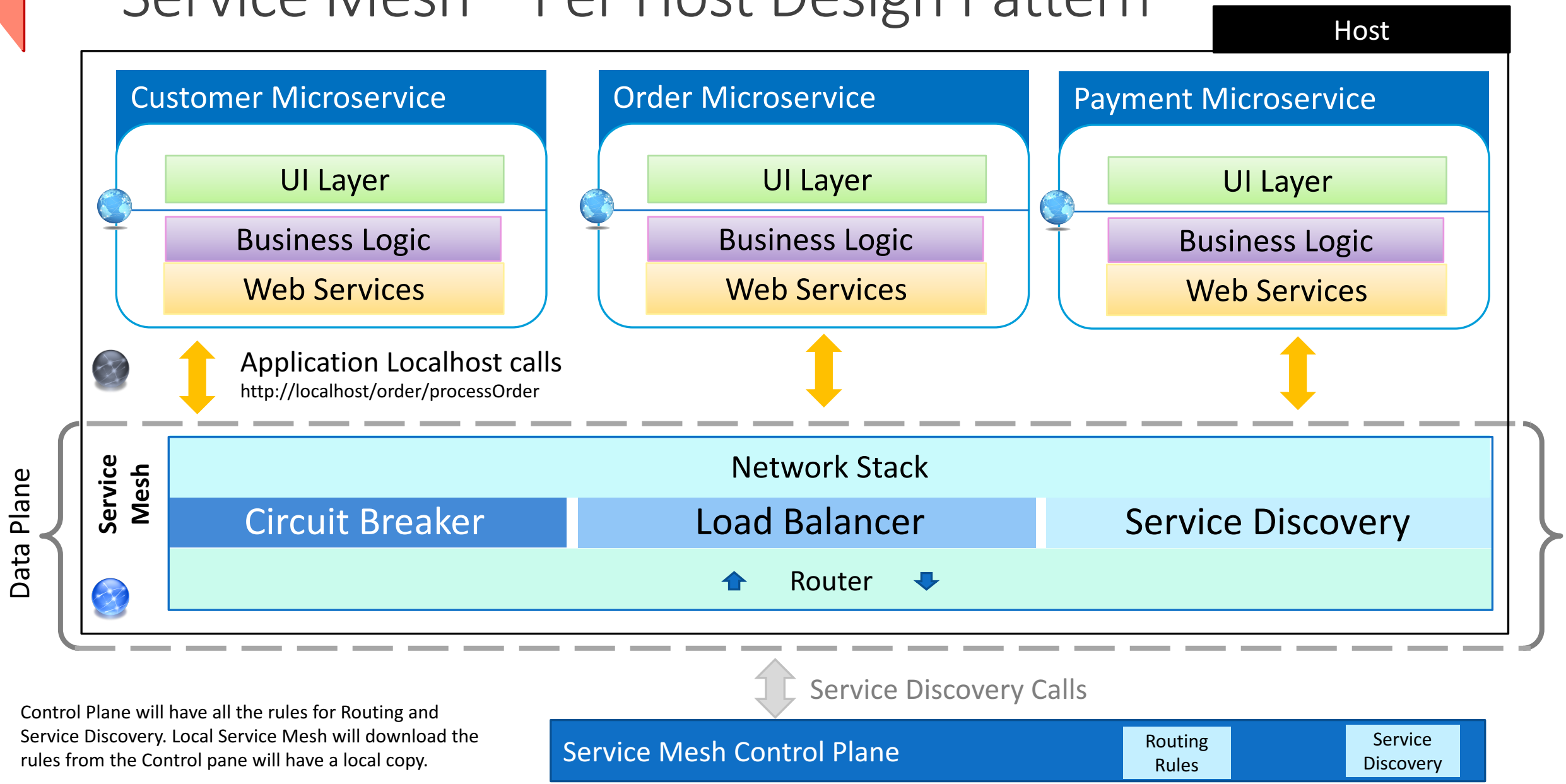
CB – Circuit Breaker
LB – Load Balancer
SD – Service Discovery

Data Plane



Control Plane will have all the rules for Routing and Service Discovery. Local Service Mesh will download the rules from the Control pane will have a local copy.

Service Mesh – Per Host Design Pattern



Control Plane will have all the rules for Routing and Service Discovery. Local Service Mesh will download the rules from the Control pane will have a local copy.

Service Mesh Software Features

Traffic Controls

- Load Balancing
  - Latency based Routing
- Service Discovery
- Access Control
- Per Request Routing
  - Shadowing
  - Fault Injection

Observability

- Request Volumes
- Success Rates
- Latency Logs
- Distributed Tracing

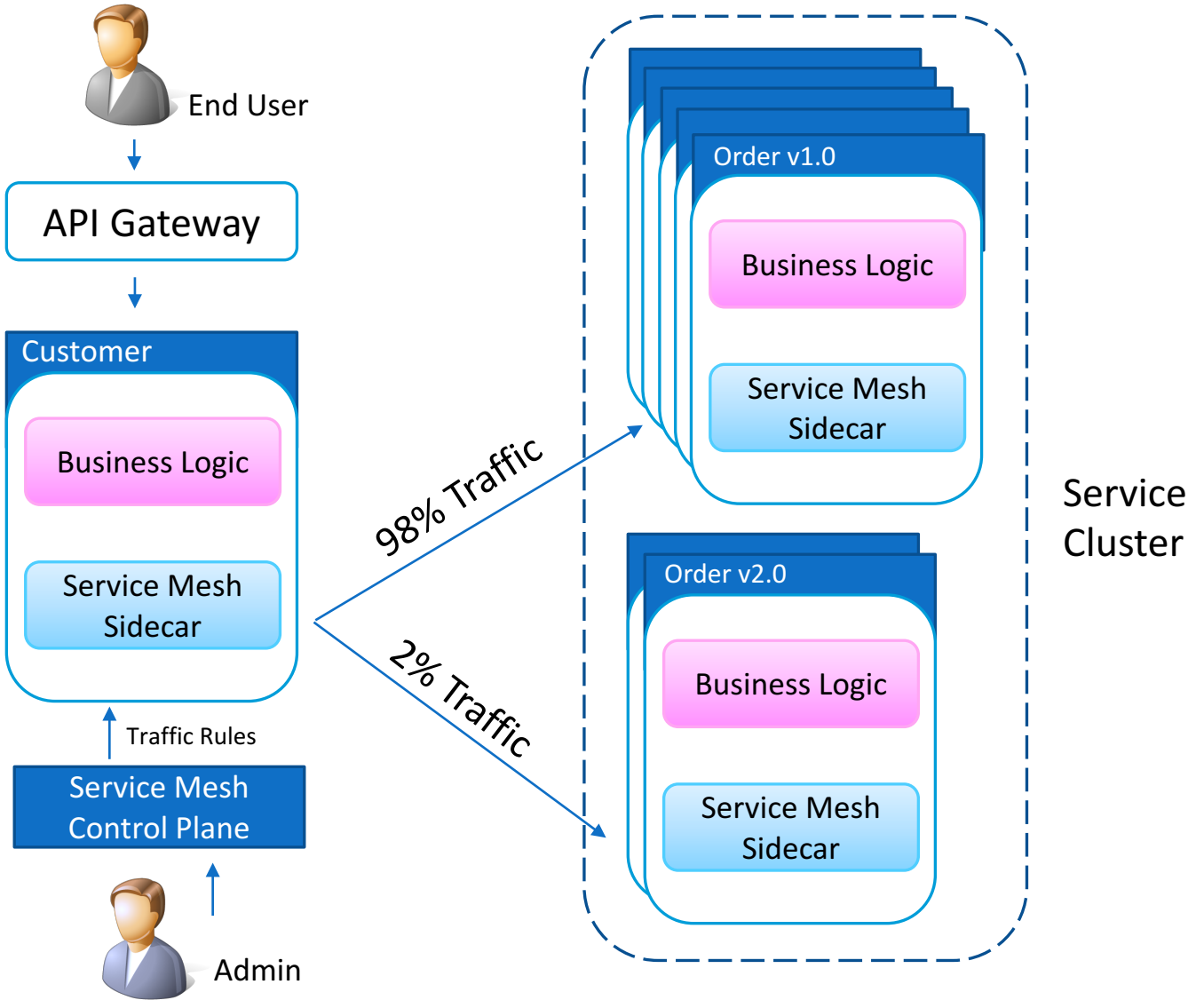
Reliability

- Health Checks
- Timeouts
- Circuit Breakers
- Retries

Service Mesh – Traffic Control

Traffic Control rules can be applied for

- different Microservices versions
- Re Routing the request to debugging system to analyze the problem in real time.
- Smooth migration path



2.8

Istio

LinkerD

Service Mesh – Open Source Infrastructures

https://istio.io/docs/concepts/what-is-istio/overview.html.

https://www.envoyproxy.io/docs/envoy/latest/intro/what_is_envoy

Libraries

Stubby + GSLB + GFE + Dapper

Finagle + Thrift + ZooKeeper + Zipkin

HTTP/JSON + SmartStack + ZooKeeper + Charon / Dyno

Karyon + HTTP/JSON or RxNetty RPC + Eureka + Hysterix + Ribbon + Zuul

Distributed Replicated Log

3

Apache Kafka

1. Kafka Features
    1. Core Concepts
    2. Producers, Consumers and Consumer Groups
    3. Kafka API
    4. Traditional Queue / Pub-Sub Vs. Kafka
2. Kafka Topic & Durability
    1. Anatomy of Topic
    2. Partition Log Segment
    3. Cluster – Topic and Partitions
    4. Record Commit Process
    5. Consumer Access & Retention Policy
    6. Replication

Apache Kafka

3. Kafka Records / Messages
    1. Kafka Record v1
    2. Kafka Record v2
    3. Kafka Record Batch
4. Kafka Operations
    1. Kafka Setup
    2. Kafka Producer / Consumer API
    3. Protocol Buffer v3.0
5. Kafka Performance
    1. LinkedIn Kafka Cluster
    2. Uber Kafka Cluster

Kafka Core Concepts

Publish & Subscribe

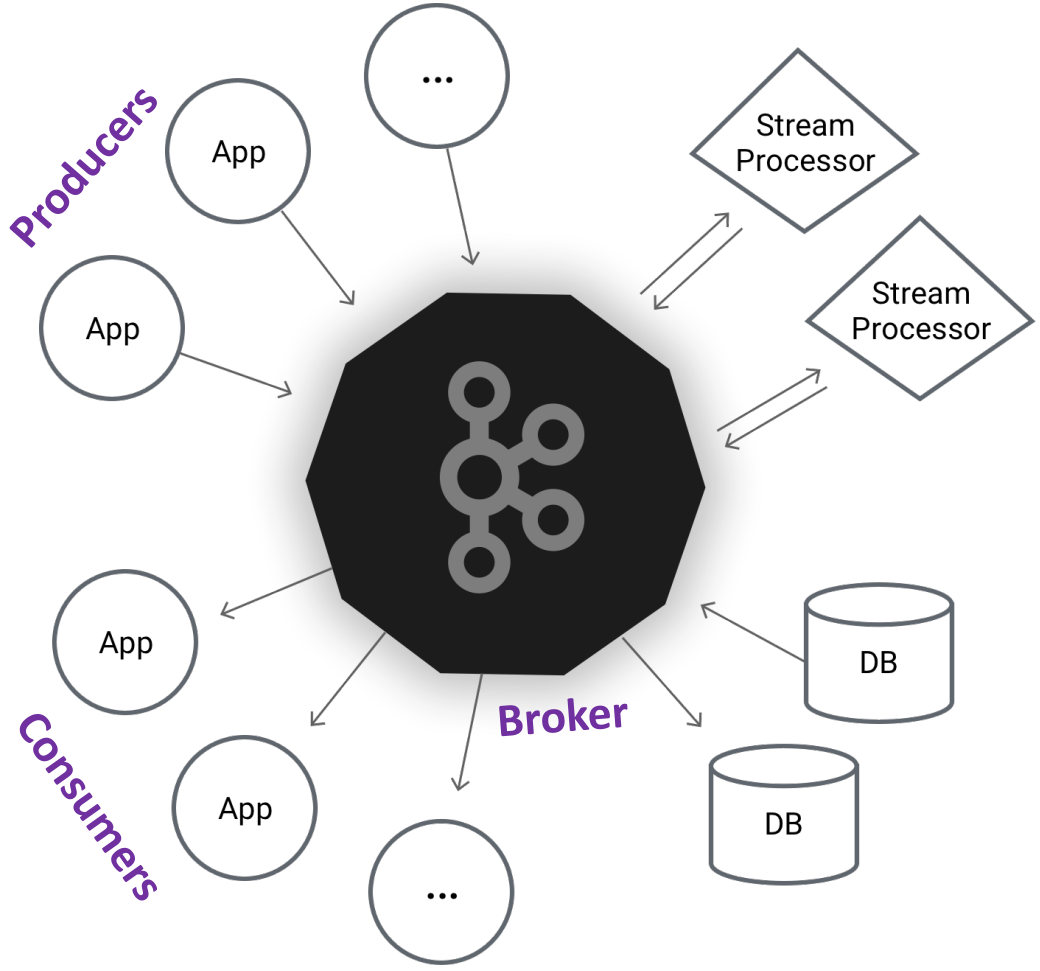
Read and write streams of data like a messaging system

Process

Write scalable stream processing apps that react to events in real-time.

Store

Store streams of data safely in a distributed, replicated, fault tolerant cluster.



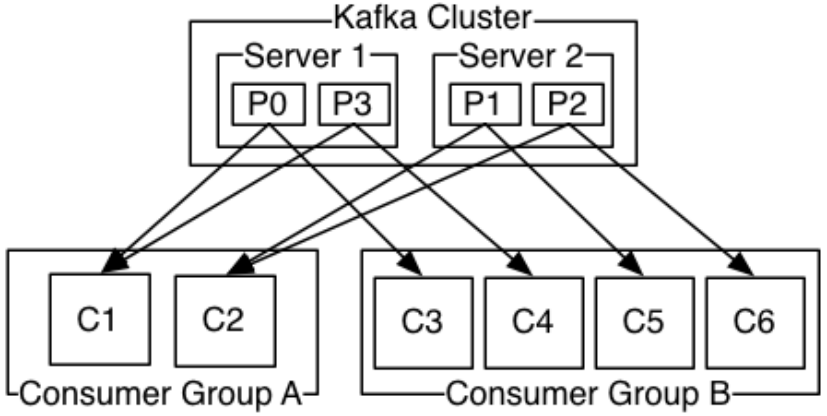
Producers, Consumers and Consumer Groups

Producers

- Producers publish data to the topics of their choice.
- The producer is responsible for choosing which record to assign to which partition within the topic.
- ***This can be done in a round-robin fashion simply to balance load or it can be done according to some semantic partition function (say based on some key in the record).***

Consumers

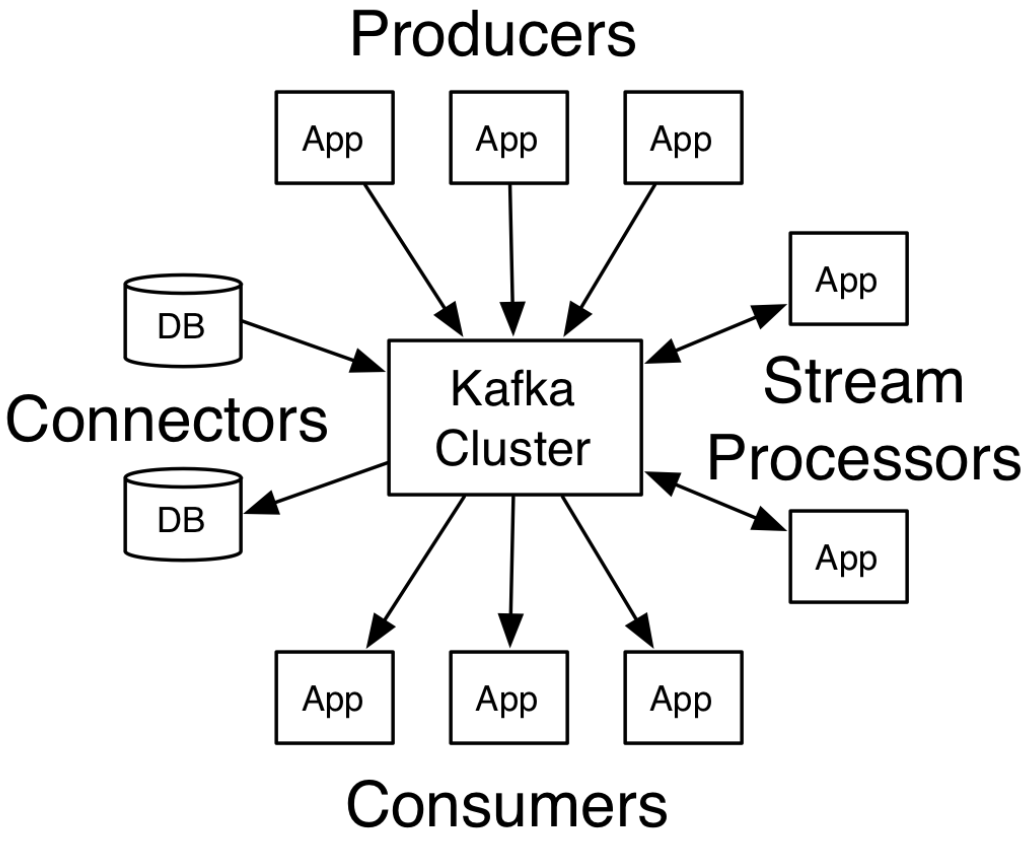
- Consumers label themselves with a *consumer group* name, and each record published to a topic is delivered to one consumer instance within each subscribing consumer group. Consumer instances can be in separate processes or on separate machines.
- If all the consumer instances have the same consumer group, then the records will effectively be load balanced over the consumer instances.
- If all the consumer instances have different consumer groups, then each record will be broadcast to all the consumer processes.



Source: https://kafka.apache.org/documentation/#api

Kafka APIs

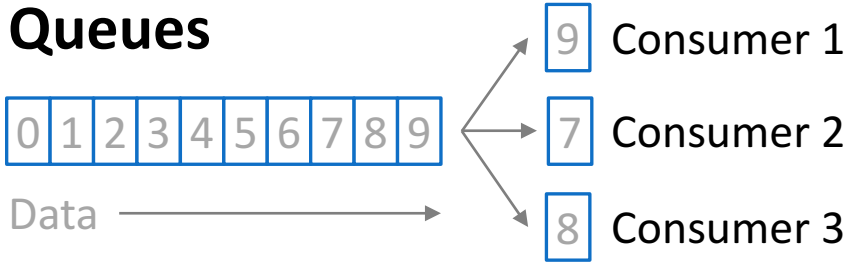
- The Producer API allows an application to publish a stream of records to one or more Kafka topics.
- The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them.
- The Streams API allows an ***application to act as a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics,*** effectively transforming the input streams to output streams.
- The Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, ***a connector to a relational database might capture every change to a table.***



Source : https://kafka.apache.org/documentation/#gettingStarted

Traditional Queue / Pub-Sub Vs. Kafka

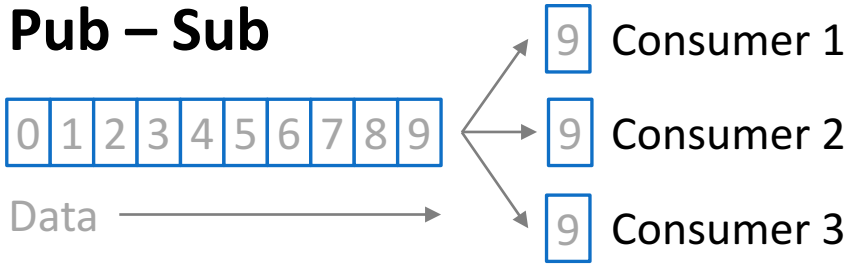
Queues



Pros: Data can be partitioned for scalability for parallel processing by same type of consumers

Cons: **Queues are not multi subscribers**. Once a Consumer reads the data, its gone from the queue. Ordering of records will be lost in asynchronous parallel processing.

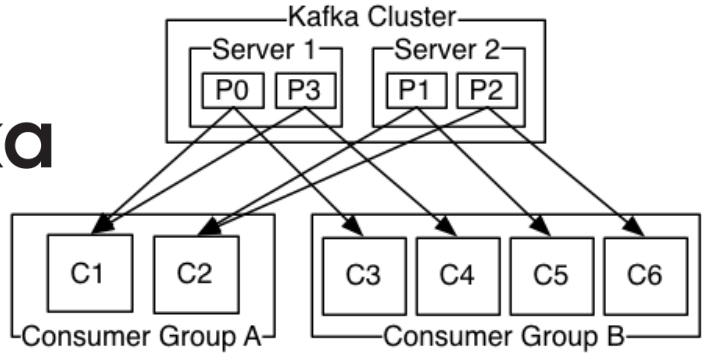
Pub – Sub



Pros: Multiple subscribers can get the same data.

Cons: **Scaling is difficult as every message goes to every subscriber.**

kafka



Kafka generalizes these two concepts.

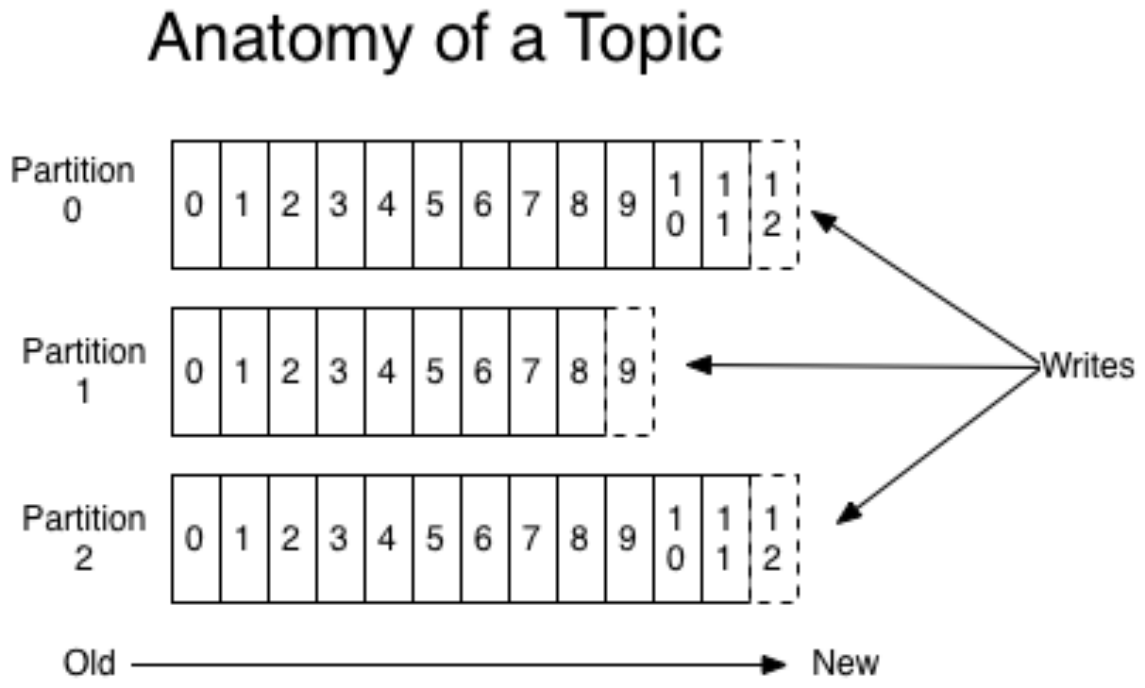
- ✓ As with a queue the consumer group allows you to divide up processing over a collection of processes (the members of the consumer group).
- ✓ As with publish-subscribe, Kafka allows you to broadcast messages to multiple consumer groups.

Kafka Topic and Durability

1. Anatomy of Topic
2. Partition Log Segment
3. Cluster – Topic and Partitions
4. Record Commit Process
5. Consumer Access & Retention Policy
6. Replication

Anatomy of a Topic

- A **Topic** is a category or feed name to which records are published.
- **Topics** in Kafka are always multi subscriber.
- That is, a **Topic** can have zero, one, or many consumers that subscribe to the data written to it.
- Each **Partition** *is an ordered, immutable sequence of records that is continually appended to—a structured commit log.*
- **A Partition is nothing but a directory of Log Files**

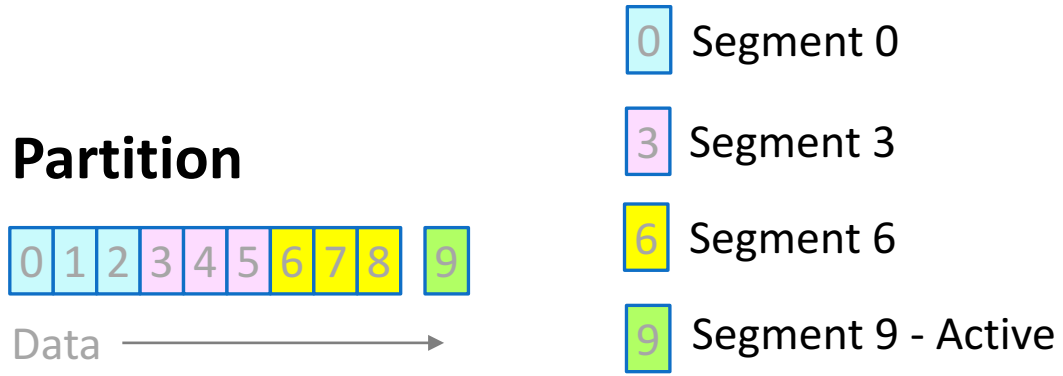


Source : https://kafka.apache.org/intro

- The records in the partitions are each assigned a *sequential id number called the offset that uniquely identifies each record within the partition*.

Partition Log Segment

- **Partition** (Kafka's Storage unit) is **Directory of Log Files**.
- A partition cannot be split across multiple brokers or even multiple disks
- **Partitions are split into Segments**
- Segments are two files: **000.log & 000.index**
- **Segments are named by their base offset**. The base offset of a segment is an offset greater than offsets in previous segments and less than or equal to offsets in that segment.
- **Indexes store offsets relative to its segments base offset**
- Indexes **map each offset to their message position** in the log and they are used to look up messages.
- **Purging of data is based on oldest segment and one segment at a time.**




```
$ tree kafka-logs | head -n 6
kafka-logs
 |—— SigmaHawk-2
 |     |—— 00000000006109871597.index
 |     |—— 00000000006109871597.log
 |     |—— 00000000007306321253.index
 |     |—— 00000000007306321253.log
```

Topic / Partition

Segment 1

Segment 2

0000.index

Rel.Offset, Position

0	0
1	7
2	11

0000.log

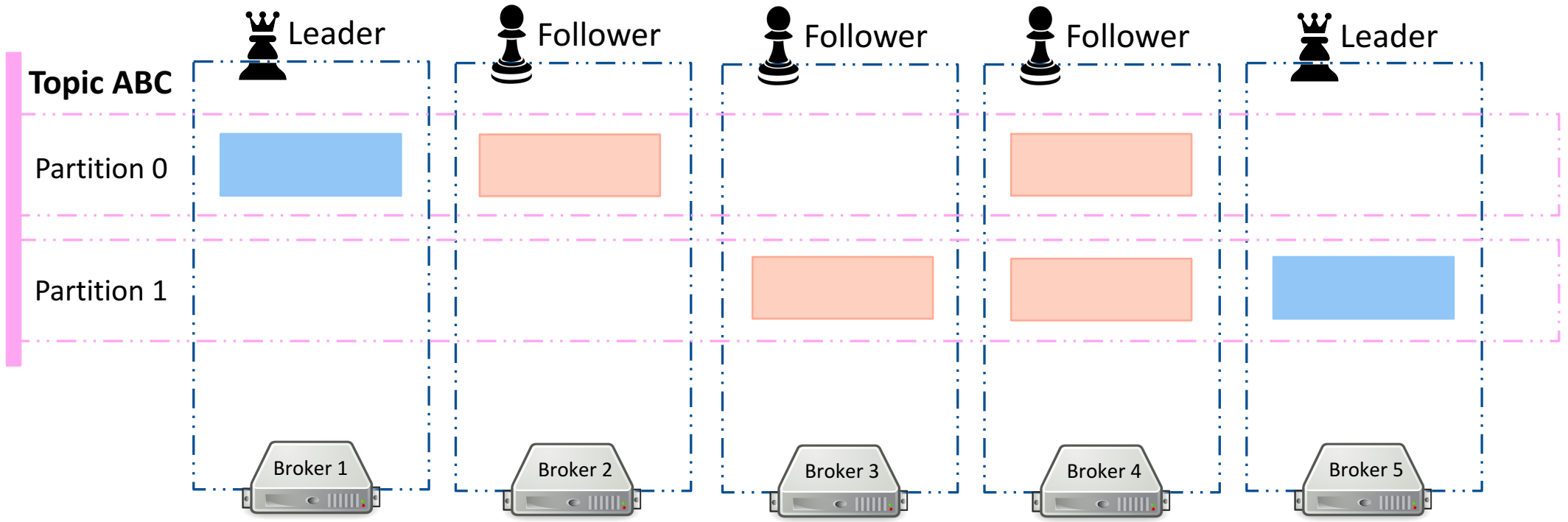
Offset, Position, Size, Payload

0	0	7	ABCDEFG
1	7	4	ABCD
2	11	9	ABCDEFGIJ

4 Bytes 4 Bytes

Kafka Cluster – Topics & Partitions

Source : https://kafka.apache.org/intro

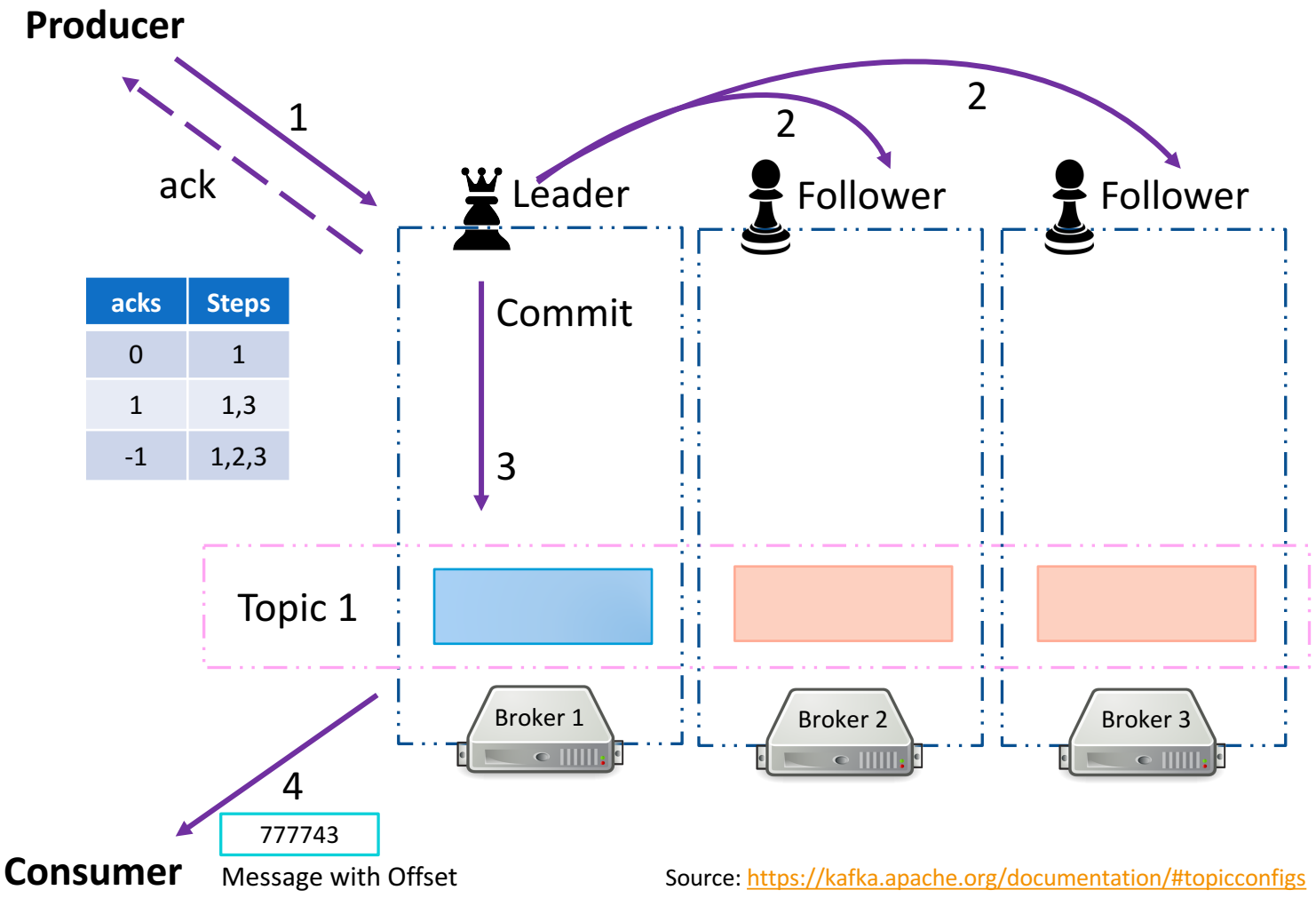


- The ***partitions of the log are distributed over the servers*** in the Kafka cluster with each server handling data and requests for a share of the partitions.
- Each **partition has one server which acts as the "leader"** and zero or more servers which act as "followers".
- Each **server acts as a leader for some of its partitions and a follower for others** so load is well balanced within the cluster.

Record Commit Process

Data Durability

From Kafka v0.8.0 onwards



acks	Steps
0	1
1	1,3
-1	1,2,3

Source: https://kafka.apache.org/documentation/#topicconfigs

- Each **partition is replicated** across a configurable number of servers for fault tolerance.
- The **leader handles all read and write requests for the partition** while the followers passively replicate the leader.
- If the **leader fails, one of the followers will automatically become the new leader.**

Producer Configuration

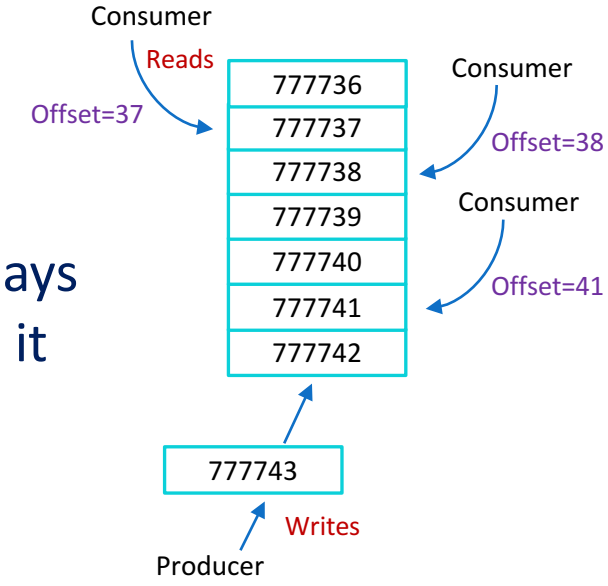
acks	Acknowledgement Description
0	**If set to zero then the producer will NOT wait for any acknowledgment from the server at all.** The record will be immediately added to the socket buffer and considered sent. No guarantee can be made that the server has received the record in this case, and the retries configuration will not take effect (as the client won't generally know of any failures). The offset given back for each record will always be set to -1.
1	**This will mean the leader will write the record to its local log but will respond without awaiting full acknowledgement from all followers.** In this case should the leader fail immediately after acknowledging the record but before the followers have replicated it then the record will be lost.
All / -1	**This means the leader will wait for the full set of in-sync replicas to acknowledge the record.** This guarantees that the record will not be lost as long as at least one in-sync replica remains alive. This is the strongest available guarantee. This is equivalent to the acks=-1 setting.

Consumer Access & Data Retention

- The Kafka cluster retains all published records—whether or not they have been consumed—using a configurable retention period

- For example, if the retention policy is set to 2 days, then for the two days after a record is published, it is available for consumption, after which it will be discarded to free up space.

- **Kafka's performance is effectively constant with respect to data size so storing data for a long time is not a problem.**



- Producers **Push Data**
- Consumers **Poll Data**

- Only *metadata retained on a per-consumer basis is the offset or position of that consumer in the log*. **This offset is controlled by the consumer**: normally a consumer will advance its offset linearly as it reads records, but, in fact, since the position is controlled by the consumer it can consume records in any order it likes.

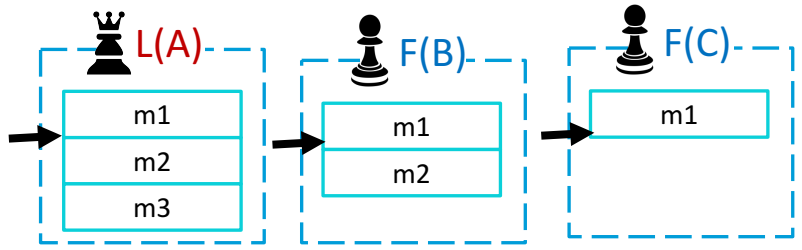
Source : https://kafka.apache.org/intro

Replication

- Instead of majority vote, Kafka dynamically **maintains a set of in-sync replicas (ISR)** that are caught-up to the leader.
- **Only members of this set are eligible for election as leader**.
- A write to a Kafka partition is **not considered committed until *all* in-sync replicas have received the write**.
- This ISR set is persisted to ZooKeeper whenever it changes. Because of this, any replica in the ISR is eligible to be elected leader.

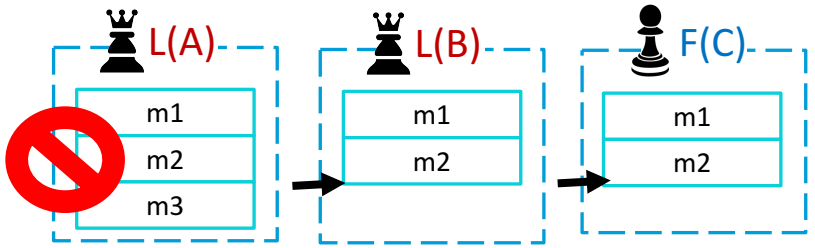
1

ISR = (A, B, C)
Leader A commits Message m1. Message m2 & m3 not yet committed.



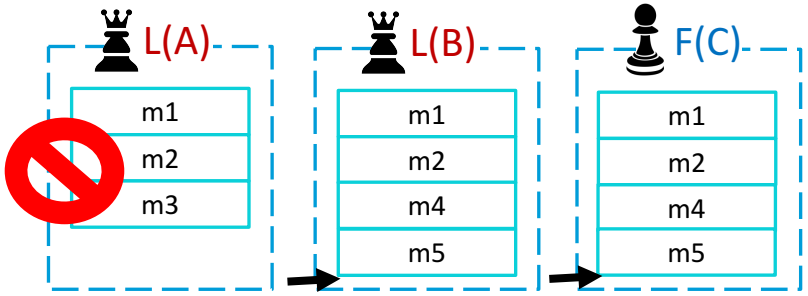
2

ISR = (B,C)
A fails and B is the new Leader. B commits m2



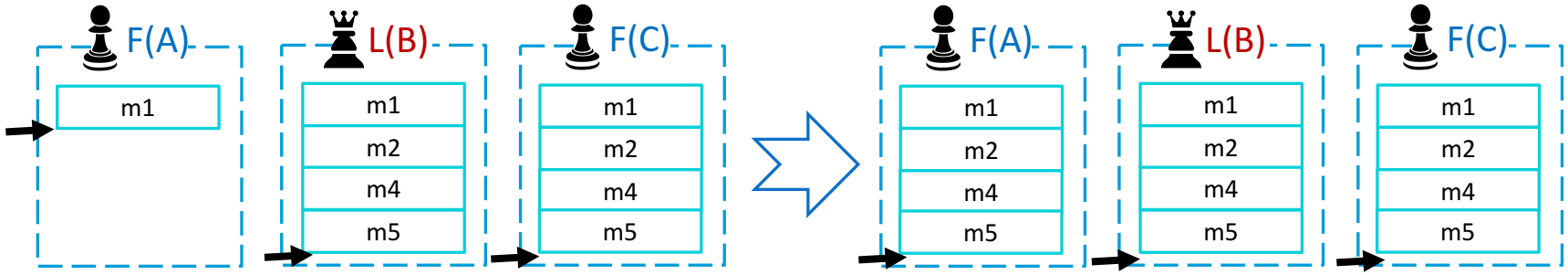
3

ISR = (B,C)
B commits new messages m4 and m5



4

ISR = (A, B,C)
A comes back, restores to last commit and catches up to latest messages.

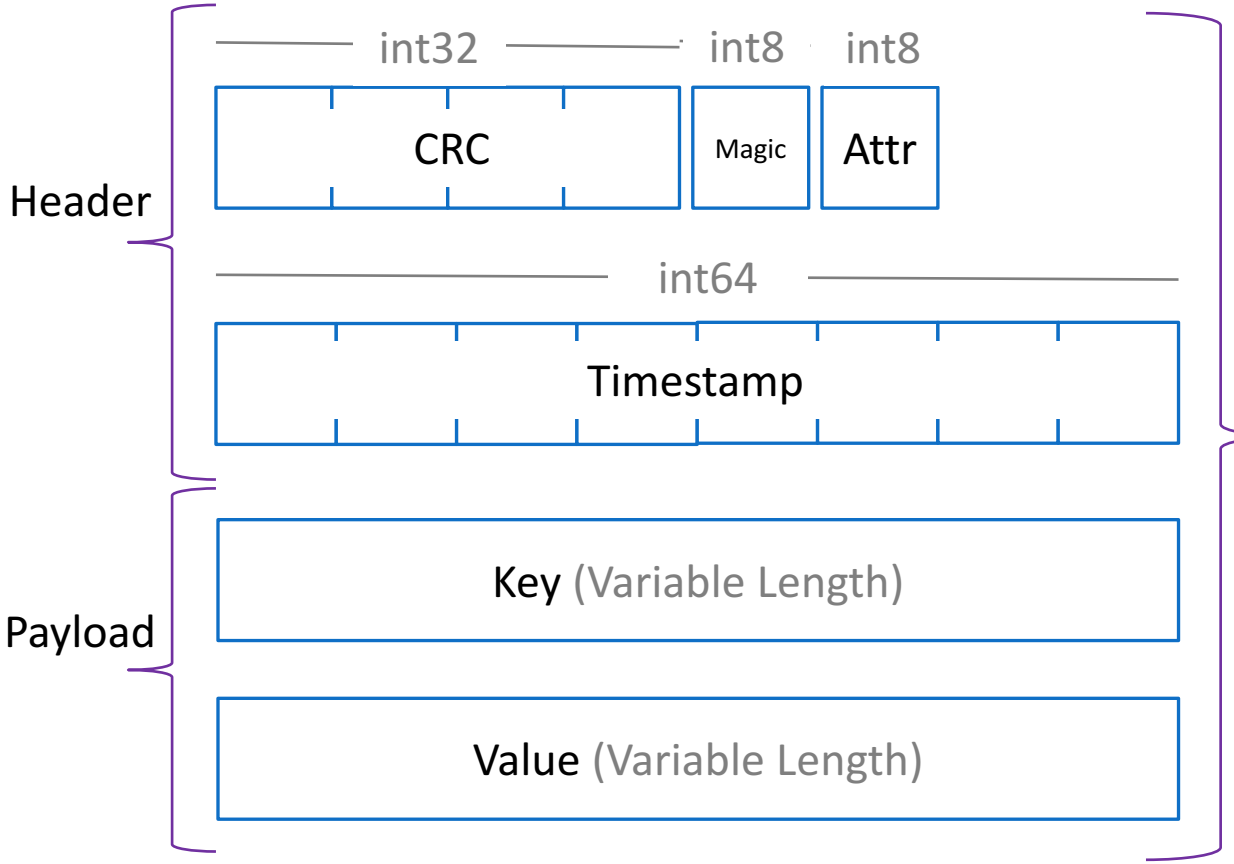


Kafka Data Structures

1. Kafka Record v1
2. Kafka Record v2
3. Kafka Record Batch

Kafka Record / Message Structure

v1 (Supported since 0.10.0)



Field	Description
CRC	The CRC is the CRC32 of the remainder of the message bytes. This is used to check the integrity of the message on the broker and consumer.
Magic Byte	This is a version id used to allow backwards compatible evolution of the message binary format**. The current value is 2.**
Attributes	Bit 0-2 Compression Codec 0 No Compression 1 Gzip Compression 2 Snappy Compression Bit 3 Timestamp Type: 0 for Create Time Stamp, 1 for Log Append Time Stamp Bit. 4 is Transactional (0 means Transactional) Bit 5 is Control Batch (0 means Control Batch) Bit >5. Un used
Timestamp	This is the timestamp of the message. The timestamp type is indicated in the attributes. Unit is milliseconds since beginning of the epoch (midnight Jan 1, 1970 (UTC)).
Key	The key is an optional message key that was used for partition assignment. The key can be null.
Value	The value is the actual message contents as an opaque byte array. Kafka supports recursive messages in which case this may itself contain a message set. The message can be null.

Source: https://kafka.apache.org/documentation/#messages

Kafka Record Structure

v2 (Supported since 0.11.0)

Record

Length (varint)	Attr (int8)	Timestamp Delta (varint)
Offset Delta (varint)		Key Length (varint)
Key (varint)		Value Length (varint)
Value (varint)		Headers (Header Array)

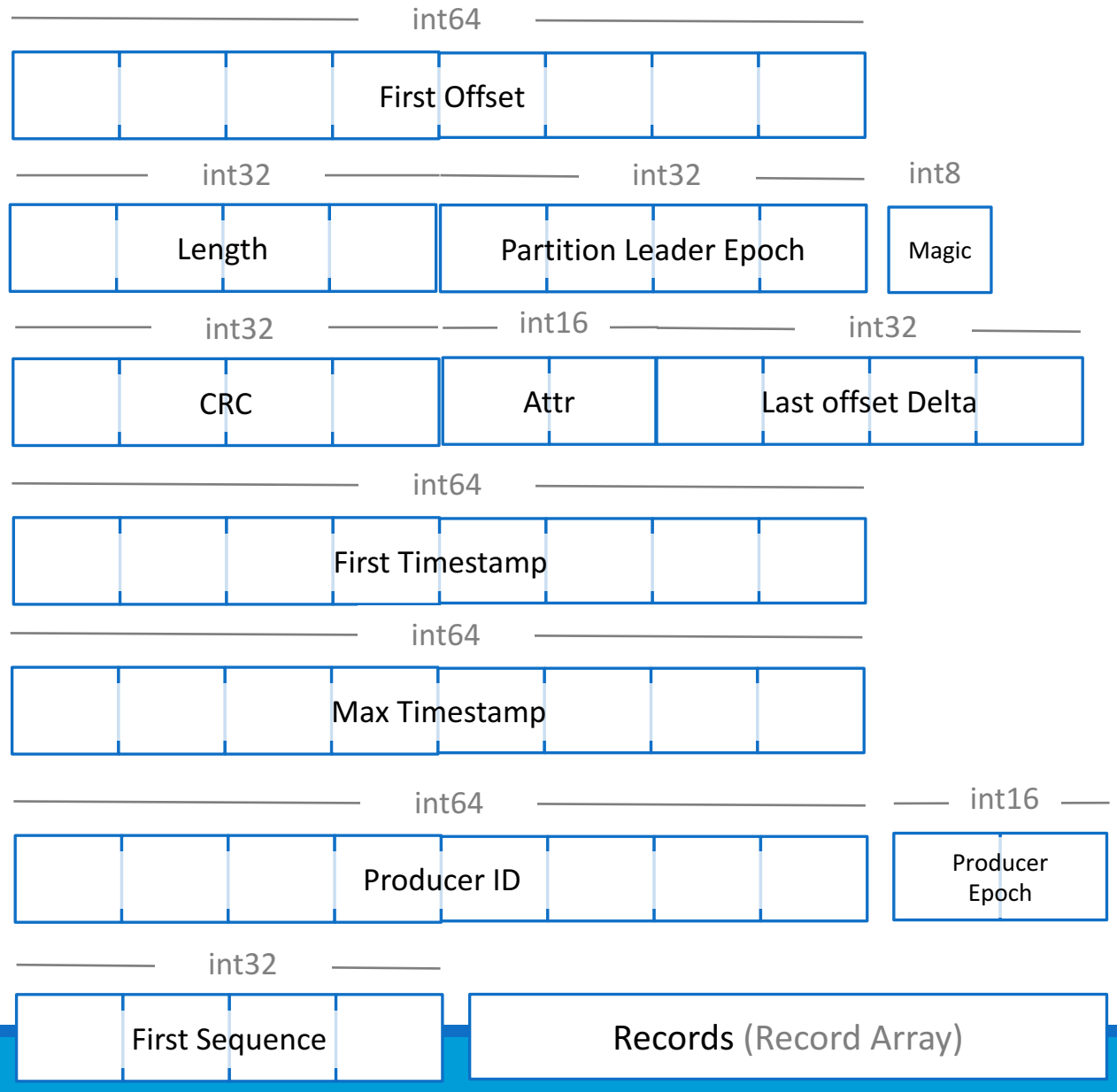
Header

Header Key Length (varint)	Header Key (varint)
Header Value Length (varint)	Header Value (varint)

- In Kafka 0.11, the structure of the 'Message Set' and 'Message' were significantly changed.
- **A 'Message Set' is now called a 'Record Batch', which contains one or more 'Records' (and not 'Messages').**
- The recursive nature of the previous versions of the message format was eliminated in favor of a flat structure.
- When compression is enabled, the Record Batch header remains uncompressed, but the Records are compressed together.
- Multiple fields in the 'Record' are varint encoded, which leads to significant space savings for larger batches.

Kafka Record Batch Structure

v2 (Supported since 0.11.0)



Field	Description
First Offset	Denotes the **first offset in the Record Batch**. The **'offset Delta**' of each Record in the batch would be be **computed relative to this First Offset**.
Partition Leader Epoch	this is set by the broker upon receipt of a produce request and is used **to ensure no loss of data when there are leader changes with log truncation.**
Attributes	The fifth lowest bit indicates whether the Record Batch is part of a transaction or not. 0 indicates that the Record Batch is not transactional, while 1 indicates that it is. (since 0.11.0.0)
Last Offset Delta	**The offset of the last message in the Record Batch**. This is used by the broker to ensure correct behavior even when Records within a batch are compacted out.
First Timestamp	The timestamp of the first Record in the batch. The timestamp of each Record in the Record Batch is its **'Timestamp Delta' + 'First Timestamp'.**
Max Timestamp	**The timestamp of the last Record in the batch.** This is used by the broker to ensure the correct behavior even when Records within the batch are compacted out.
Producer ID	This is the **broker assigned producer Id** received by the 'Init Producer Id' request.
Producer Epoch	This is the **broker assigned producer Epoch** received by the 'Init Producer Id' request.
First Sequence	This is the producer assigned sequence number which is used by the broker to de-duplicate messages. The sequence number for each Record in **the Record Batch is its Offset Delta + First Sequence.**

Kafka Operations

Kafka Quick Setup & Demo

1. install the most recent version from Kafka download page
2. Extract the binaries into a /…./Softwares/kafka folder. For the current version it's kafka_2.11-1.0.0.0.
3. Change your current directory to point to the new folder.
4. Start the **Zookeeper server** by executing the command:
   bin/zookeeper-server-start.sh config/zookeeper.properties.
5. Start the **Kafka server** by executing the command:
   bin/kafka-server-start.sh config/server.properties.
6. Create a **Test topic** that you can use for testing:
   bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic KafkaSigmaTest
7. Start a simple console **Consumer** that can consume messages published to a given topic, such as KafkaSigmaTest:
   bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic KafkaSigmaTest --from-beginning.
8. Start up a simple **Producer** console that can publish messages to the test topic:
   bin/kafka-console-producer.sh --broker-list localhost:9092 --topic KafkaSigmaTest
9. Try typing one or two messages into the producer console. Your messages should show in the consumer console.

Kafka Producer (Java)

Kafka Producer Configuration

```
public class KafkaConfig {

    private final static String BOOTSTRAP_SERVERS =
                                "localhost:9093,localhost:9094,"
                              + "localhost:9095,localhost:9096";

    /**
     * Normal Producers
     *
     * @return
     */
    public static Producer<Long, String> createProducer() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            BOOTSTRAP_SERVERS);
        props.put(ProducerConfig.CLIENT_ID_CONFIG,
            "KafkaExampleProducer");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            LongSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName());
        props.put("acks", "all");
        props.put("retries", 0);
        props.put("batch.size", 16384);
        props.put("linger.ms", 1);
        props.put("buffer.memory", 33554432);
        return new KafkaProducer<>(props);
    }
```

Kafka Producer Synchronous

```
16 /**
17  *
18  * @author arafkarsh
19  *
20  */
21 public class KafkaProducerSyncExample {
22
23     public static void main (String[] args) throws InterruptedException, ExecutionException {
24
25         producerSyncSendExample("KafkaSigma2Sync", 7);
26     }
27
29      * Synchronous Kafka Producer
36     public static void producerSyncSendExample(final String _topic, final
37             int _sendMessageCount) throws InterruptedException, ExecutionException {
38         final Producer<Long, String> producer = KafkaConfig.createProducer();
39         long time = System.currentTimeMillis();
40
41           try {
42               for (long index = time; index < time + _sendMessageCount; index++) {
43                 createKafkaRecord(producer, _topic, index, time, "KafkaSyncRecords|" + index);
44               }
45           } finally {
46               producer.flush();
47               producer.close();
48           }
49     }
50
52      * Create a Kafka Record
61     private static void createKafkaRecord(Producer<Long, String> _producer, String _topic,
62             long _index, long _time, String _record) throws InterruptedException, ExecutionException {
63         final ProducerRecord<Long, String> record =
64                 new ProducerRecord<>( topic,  index,  record);
65         RecordMetadata metadata = _producer.send(record).get();
66         long elapsedTime = System.currentTimeMillis() - _time;
67         System.out.printf("Sent record(key=%s value=%s) " +
68               "meta(partition=%d, offset=%d) time=%d\n", record.key(),
69               record.value(), metadata.partition(), metadata.offset(), elapsedTime);
70     }
71 }
```

Kafka Consumer (Java)

Kafka Consumer Configuration

```java
public static KafkaConsumer<String, String> createConsumer(int _fetchSize) {
    Properties props = new Properties();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
    props.put("key.deserializer",
        "org.apache.kafka.common.serialization.StringDeserializer");
    props.put("value.deserializer",
        "org.apache.kafka.common.serialization.StringDeserializer");
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "sigmaGroup");
    props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");
    props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "1000");
    props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, _fetchSize+"");
    KafkaConsumer<String, String> consumer =
        new KafkaConsumer<String, String>(props);
    return consumer;
}
```

Kafka Consumer

```java
public class KafkaConsumerExample {

    public static void main (String[] args) throws InterruptedException,
                ExecutionException {

        String topic1 = new String("KafkaSigma2Sync");
        // String topic2 = new String("KafkaSigma2Async");
        // String topic3 = new String("KafkaSigma2Txn");

        consumerExampleSync(topic1, 2);
    }
    /**
     *
     * @param _topic
     * @param _fetchSize
     */
    public static void consumerExampleSync(String _topic, int _fetchSize) {

        KafkaConsumer<String, String> consumer = KafkaConfig.createConsumer(_fetchSize);
        consumer.subscribe(Arrays.asList(_topic));
        while (true) {
            // Fetch Records
            ConsumerRecords<String, String> records = consumer.poll(100);
            for (ConsumerRecord<String, String> record : records) {
                System.out.printf("Received offset = %d, Topic = %s, partition = %d, "
                        + "value = %s%n", record.offset(), record.topic(),
                        record.partition(), record.value());
            }
            // Do Processing with the fetched Records

            // Commit the Offset value as READ for all the fetched Records
            consumer.commitSync();
        }
    }
}
```

ProtoBuf v3.0

protobuf — vi build — 106×54

...f — vi build | ...ba — -bash | ...vi build.sh | ...va — -bash | ...va — -bash | ....0 — -bash | ...os — -bash

```
#!/bin/bash
SRC_DIR=~/CloudDrive/My_WorkSpace/Eclipse/2017/FusionCodeKafka/src/com/metamagic/fusioncold/kafka/
DST_DIR=~/CloudDrive/My_WorkSpace/Eclipse/2017/FusionCodeKafka/src/

protoc -I=$SRC_DIR/protobuf/ --java_out=$DST_DIR $SRC_DIR/protobuf/Order.proto
```

protobuf — vi Order.proto — 10

...Order.proto | ...ba — -bash | ...vi build.sh | ...va — -bash

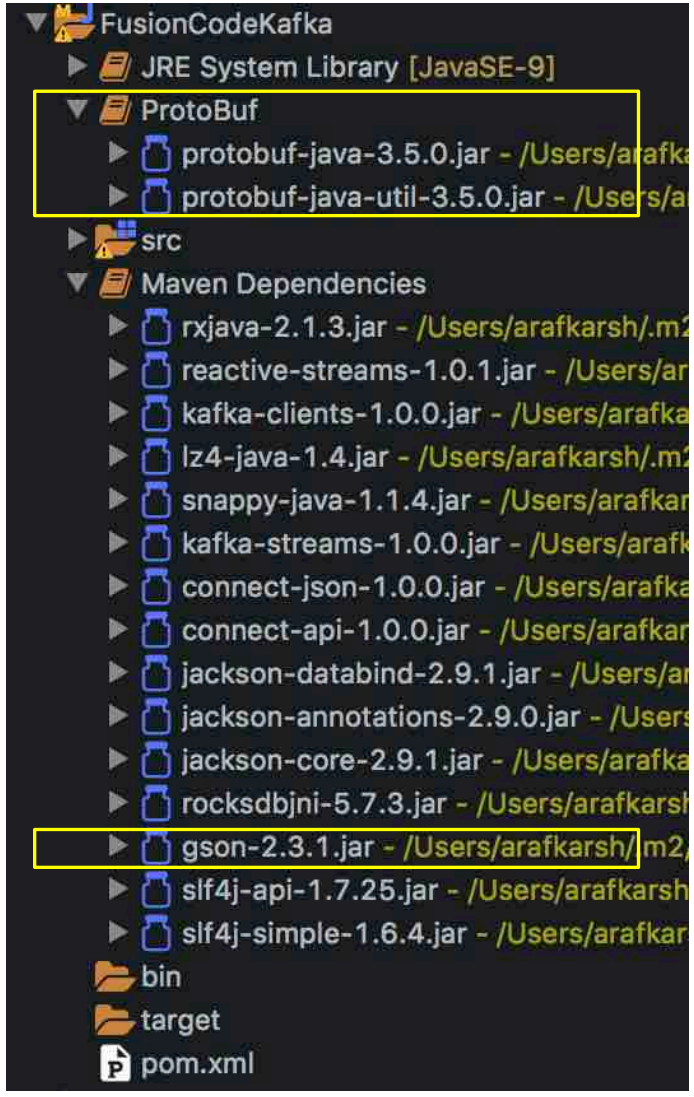
```
syntax = "proto3";

package com.metamagic.fusioncold.kafka.protos;

option java_package = "com.metamagic.fusioncold.kafka.protos";
option java_outer_classname = "OrderProtos";

message Customer {
        string customerId       = 1;
        string firstName        = 2;
        string middleName       = 3;
        string lastName         = 4;
        string salutation       = 5;

        string dateOfBirth      = 6;
        int32 gender            = 7;
}
message OrderItem {
        string orderId          = 1;
        string itemId           = 2;
        string productId        = 3;
        int32 quantity          = 4;
        double value            = 5;
        double tax              = 6;
        double totalValue       = 7;
}

message ShippingAddress {
        string shippindId       = 1;
        string address1         = 2;
        string address2         = 3;
        string city             = 4;
        string stateId          = 5;
        string countryId        = 6;
        string zipCode          = 7;
}
message Order {

        string orderId                  = 1;
        string orderLabel               = 2;
        string orderDate                = 3;

        Customer customer               = 4;
        repeated OrderItem items        = 5;
        ShippingAddress address         = 6;
}
```

ProtoBuf - Java

```java
/**
 * ProtoBuf v3.0 Examples
 */
package com.metamagic.fusioncold.kafka.examples;

import java.util.Date;

import com.metamagic.fusioncold.kafka.protos.OrderProtos.Order;
import com.metamagic.fusioncold.kafka.protos.OrderProtos.OrderItem;
import com.metamagic.fusioncold.kafka.protos.OrderProtos.Customer;
import com.metamagic.fusioncold.kafka.protos.OrderProtos.ShippingAddress;

import com.google.protobuf.InvalidProtocolBufferException;
import com.google.protobuf.Message;
import com.google.protobuf.util.JsonFormat;

```

```java
/**
 * Create the Order Object
 *
 * @param _orderId
 * @param _totalItems
 * @return
 */
public static Order buildOrderUsingProtoBuf3(int _orderId, int _totalItems) {
    Customer customer = Customer.newBuilder()
            .setCustomerId("ABCD")
            .setDateOfBirth(new Date().getTime()+"")
            .setGender(0)
            .setFirstName("John")
            .setLastName("Doe")
            .build();

    ShippingAddress shippingAddress = ShippingAddress.newBuilder()
            .setAddress1("327 Cobblestone Ln")
            .setAddress2("")
            .setCity("Edison")
            .setStateId("NJ")
            .setCountryId("USA")
            .setZipCode("08820")
            .build();

    Order order = Order.newBuilder()
            .setOrderId("XYZ1234567890")
            .setOrderDate(new Date().getTime()+"")
            .setCustomer(customer)
            .setAddress(shippingAddress)
            .addItems(OrderItem.newBuilder()
                    .setOrderId("XYZ1234567890")
                    .setItemId("1001")
                    .setProductId("SonyBravia-200xT")
                    .setQuantity(1)
                    .setValue(65000)
                    .setTax(18)
                    .build()
                    )
            .addItems(OrderItem.newBuilder()
                    .setOrderId("XYZ1234567890")
                    .setItemId("1002")
                    .setProductId("Panasonic-TY-321")
                    .setQuantity(1)
                    .setValue(42000)
                    .setTax(18)
                    .build()
                    )
            .addItems(OrderItem.newBuilder()
                    .setOrderId("XYZ1234567890")
                    .setItemId("1003")
                    .setProductId("Akkai-320GQ")
                    .setQuantity(1)
                    .setValue(85000)
                    .setTax(18)
                    .build()
                    )
            .build();

    return order;
}
```

ProtoBuf – Object to JSON & JSON to Object



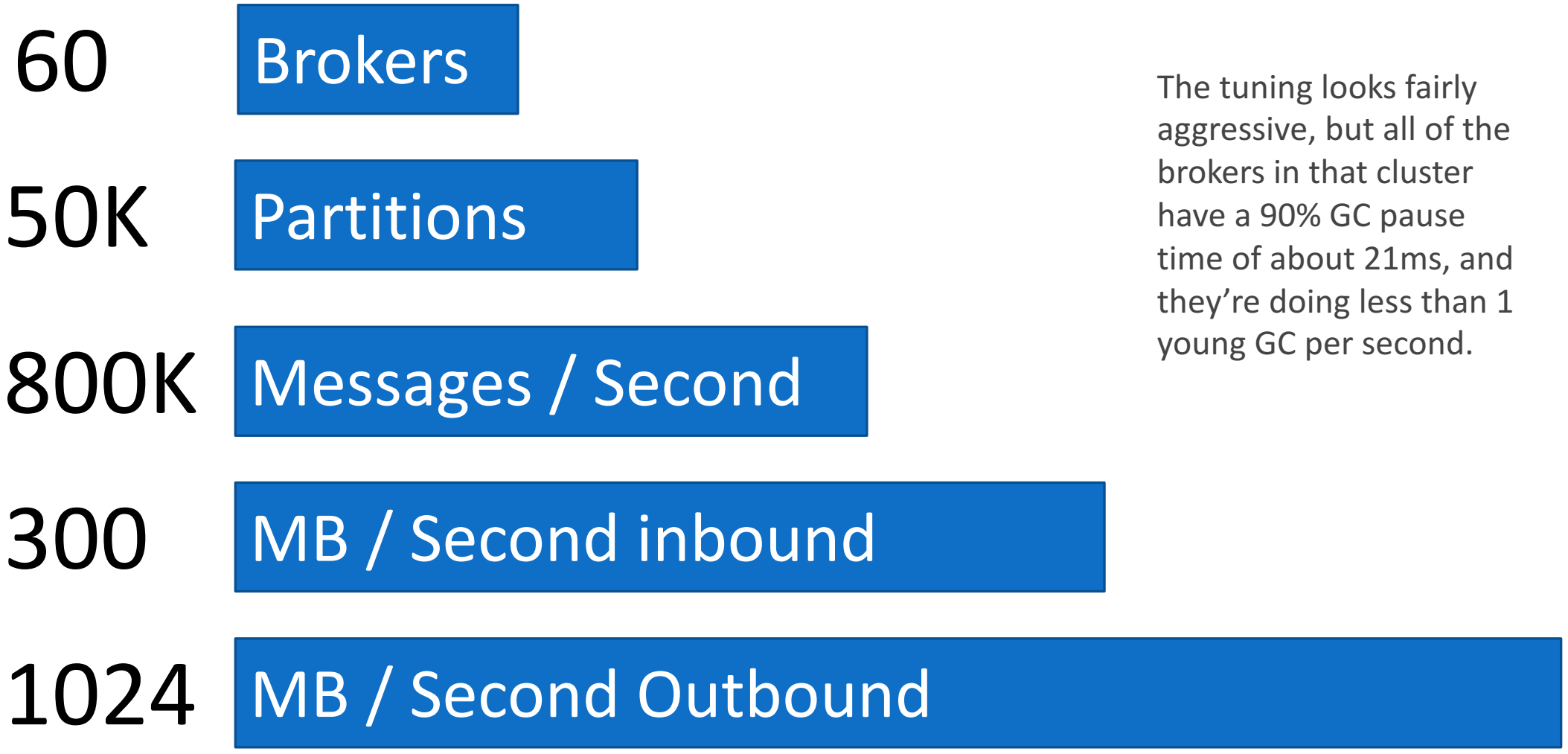

```java
public class ProtoBufExample {

    /**
     * @param args
     * @throws InvalidProtocolBufferException
     */
    public static void main(String[] args) throws InvalidProtocolBufferException {

        // Build Order Object from ProtoBuf3
        Order order1 = buildOrderUsingProtoBuf3(100, 3);
        // Creating JSON from Order Object
        String jsonOrder1 = JsonFormat.printer().print(order1);
        // Print Order JSON
        System.out.println("------------------- ORDER to JSON -----------------");
        System.out.println(jsonOrder1);
        System.out.println("------------------- ORDER to JSON -----------------");

        // Create Order Builder
        Message.Builder builder = Order.newBuilder();
        // Parse JSON to Build the Order
        JsonFormat.parser().merge(jsonOrder1, builder);
        // Build the Order Object
        Order order2 = (Order) builder.build();
        // Print the JSON Again!
        System.out.println("------------------- JSON to ORDER -----------------");
        System.out.println(JsonFormat.printer().print(order2));
        System.out.println("------------------- JSON to ORDER -----------------");
    }
```

Object to JSON

JSON to Object

Kafka Performance

LinkedIn Kafka Cluster



The tuning looks fairly aggressive, but all of the brokers in that cluster have a 90% GC pause time of about 21ms, and they're doing less than 1 young GC per second.

Uber Kafka Cluster

10K+ Topics

11M Events / Second

1PB+ Petabyte of Data

https://docs.confluent.io/current/kafka/deployment.html

References

1. Linkerd + Kubernetes
2. Installing Istio
3. Tim Perrett: Envoy with Nomad and Consul
4. NGINX Fabric Model
5. William Morgan - Linkerd:
6. Christian Posta – Envoy/Istio
7. Matt Klein – Envoy:
8. Kelsey Hightower – Istio
9. https://www.youtube.com/watch?v=s4qasWn_mFc

Thank you

Araf Karsh Hamid :
Co-Founder / CTO MetaMagic Global Inc. NJ, USA
araf.karsh@metamagic.in | araf.karsh@gmail.com
USA: +1 (973) 969-2921
India: +91.999.545.8627
Skype / LinkedIn / Twitter / Slideshare : arafkarsh
http://www.slideshare.net/arafkarsh
https://www.linkedin.com/in/arafkarsh/

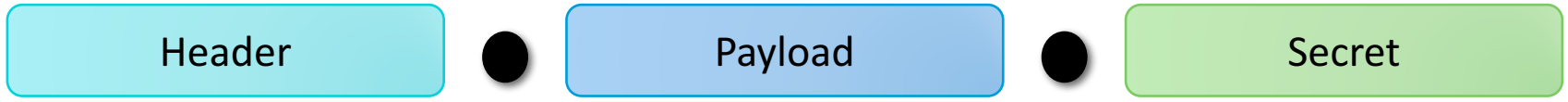
*A **Micro Service** will have its own Code Pipeline for build and deployment functionalities and it's scope will be defined by the Bounded Context focusing on the Business Capabilities and the interoperability between Micro Services will be achieved using (asynchronous) message based communication.*

4

JSON Web Token

1. JSON Web token

JSON Web Tokens : http://jwt.io



JSON Web Tokens (JWT)

Pronounced “jot”, are a standard since the information they carry is transmitted via JSON. We can read more about the draft, but that explanation isn’t the most pretty to look at.

JSON Web Tokens work across different programming languages:

JWTs work in .NET, Python, Node.JS, Java, PHP, Ruby, Go, JavaScript, and Haskell. So you can see that these can be used in many different scenarios.

JWTs are self-contained:

They will carry all the information necessary within itself. This means that a JWT will be able to transmit basic information about itself, a payload (usually user information), and a signature.

JWTs can be passed around easily:

Since JWTs are self-contained, they are perfectly used inside an HTTP header when authenticating an API. You can also pass it through the URL.

JSON Web Signature
JWS: RFC 7515

JSON Web Encryption
JWE : RFC 7516

JSON Web Key
JWK : RFC 7517

JSON Web Algorithms
JWA : RFC 7518

JSON Web Token
JWT: RFC 7519

Authentication Traditional Way

The Problems with Server Based Authentication. A few major problems arose with this method of authentication.

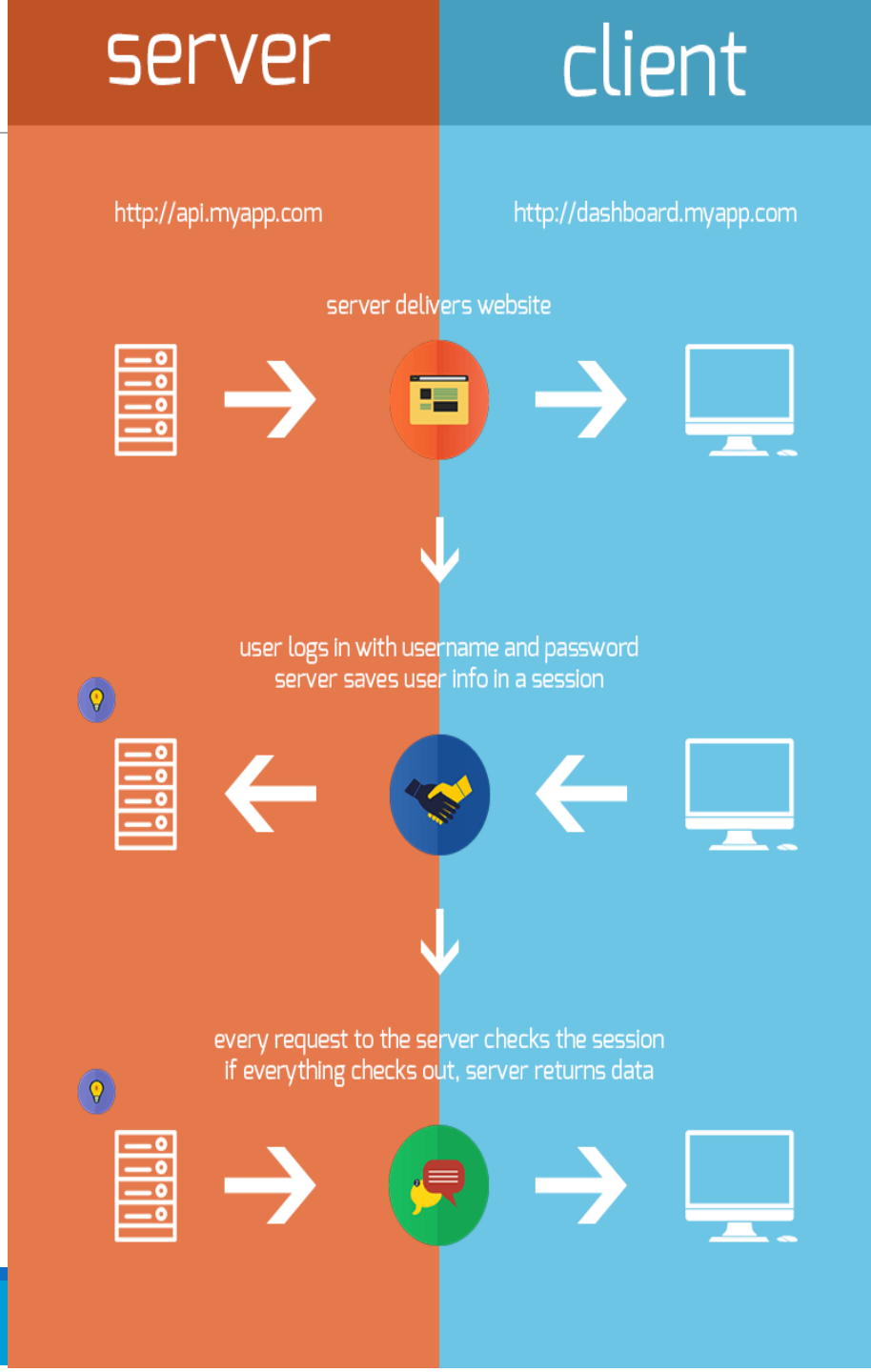
Sessions: Every time a user is authenticated, the server will need to create a record somewhere on our server. This is usually done in memory and when there are many users authenticating, the overhead on your server increases.

Scalability: Since sessions are stored in memory, this provides problems with scalability. As our cloud providers start replicating servers to handle application load, having vital information in session memory will limit our ability to scale.

CORS: As we want to expand our application to let our data be used across multiple mobile devices, we have to worry about cross-origin resource sharing (CORS). When using AJAX calls to grab resources from another domain (mobile to our API server), we could run into problems with forbidden requests.

CSRF: We will also have protection against cross-site request forgery (CSRF). Users are susceptible to CSRF attacks since they can already be authenticated with say a banking site and this could be taken advantage of when visiting other sites.

With these problems, scalability being the main one, it made sense to try a different approach.



Source: https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication

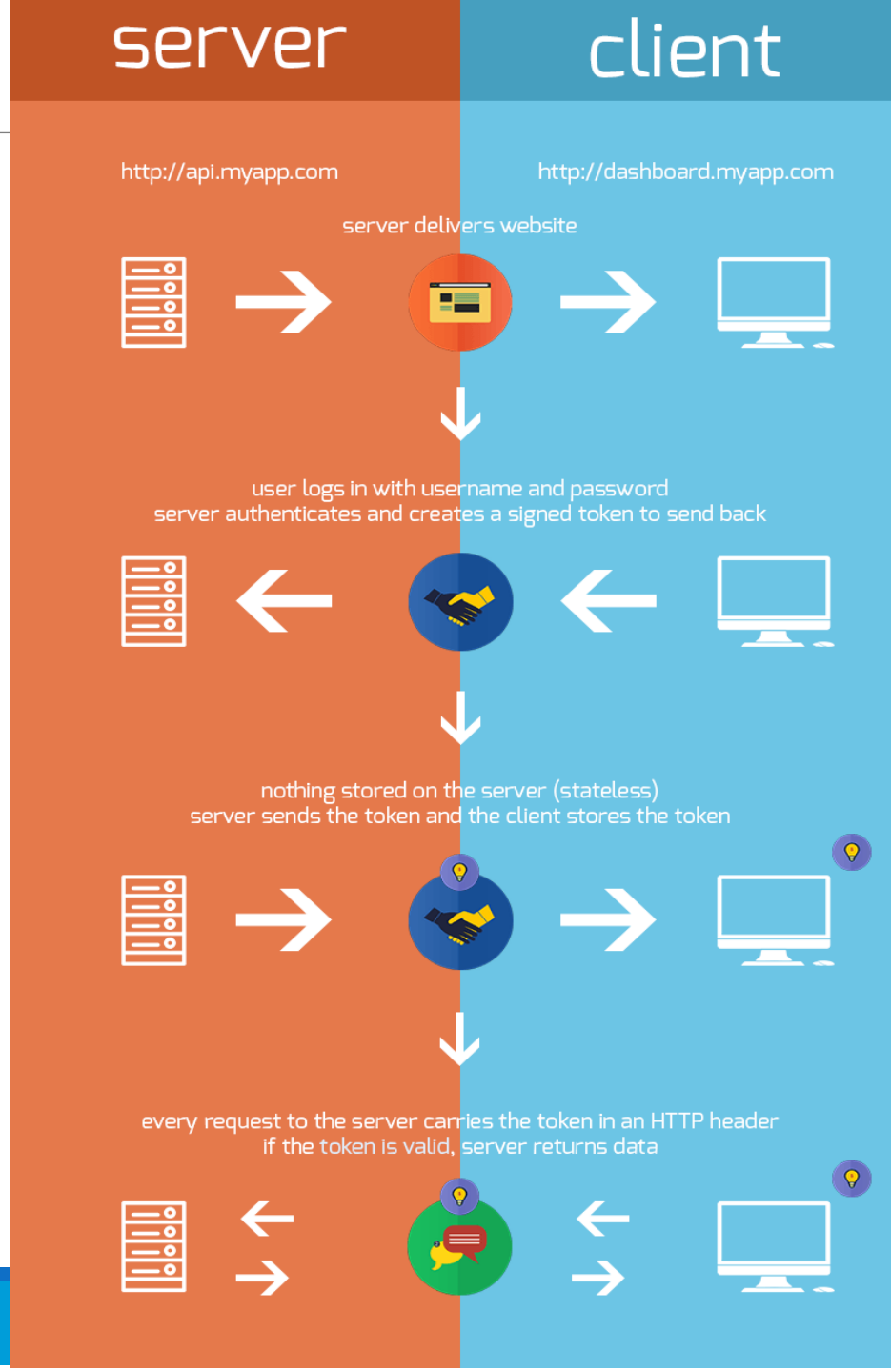
JSON Web Token: Stateless Architecture

How it Works

Token based authentication is stateless. We are not storing any information about our user on the server or in a session.

1. User Requests Access with Username / Password
2. Application validates credentials
3. Application provides a signed token to the client
4. Client stores that token and sends it along with every request
5. Server verifies token and responds with data

Every single request will require the token. This token should be sent in the HTTP header so that we keep with the idea of stateless HTTP requests. We will also need to set our server to accept requests from all domains using Access-Control-Allow-Origin: *. What's interesting about designating * in the ACAO header is that it does not allow requests to supply credentials like HTTP authentication, client-side SSL certificates, or cookies.



Source: https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication

JSON Web Token : Benefits

Stateless and Scalable

Tokens stored on client side. Completely stateless, and ready to be scaled. Our load balancers are able to pass a user along to any of our servers since there is no state or session information anywhere.

Security

The token, not a cookie, is sent on every request and since there is no cookie being sent, this helps to prevent CSRF attacks. Even if your specific implementation stores the token within a cookie on the client side, the cookie is merely a storage mechanism instead of an authentication one. There is no session based information to manipulate since we don't have a session!

Extensibility (Friend of A Friend and Permissions)

Tokens will allow us to build applications that share permissions with another. For example, we have linked random social accounts to our major ones like Facebook or Twitter.

Multiple Platforms and Domains

When the application and service expands, we will need to be providing access to all sorts of devices and applications (since the app will most definitely become popular!).

Having the App API just serve data, we can also make the design choice to serve assets from a CDN. This eliminates the issues that CORS brings up after we set a quick header configuration for our application.

Standards Based

Source: https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication

JSON Web Token : Anatomy

Header

The header carries 2 parts:

1. declaring the type, which is JWT
2. the hashing algorithm to use (HMAC SHA256 in this case)

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Payload

The payload will carry the bulk of our JWT, also called the JWT Claims. This is where we will put the information that we want to transmit and other information about our token.

```
{
  "iss": "scotch.io",
  "exp": 1300819380,
  "name": "Chris Sevilleja",
  "admin": true
}
```

```
eyJpc3MiOiJzY290Y2guaW8iLCJleHAiOjEzMDA4MTkzODAsIm
```

Secret

This signature is made up of a hash of the following components:

1. The header
2. The payload
3. Secret

The secret is the signature held by the server. This is the way that our server will be able to verify existing tokens and sign new ones.
This gives us the final part of our JWT.

```
var encodedString = base64UrlEncode(header)
          + "." + base64UrlEncode(payload);
HMACSHA256(encodedString, 'secret');
```

```
03f329983b86f7d9a9f5fef85305880101d5e302afa
```

```
eyJhbGcioiJI---.eyJpc3Mioi---.03f32998---
```

JSON Web Token : Anatomy – The Claims

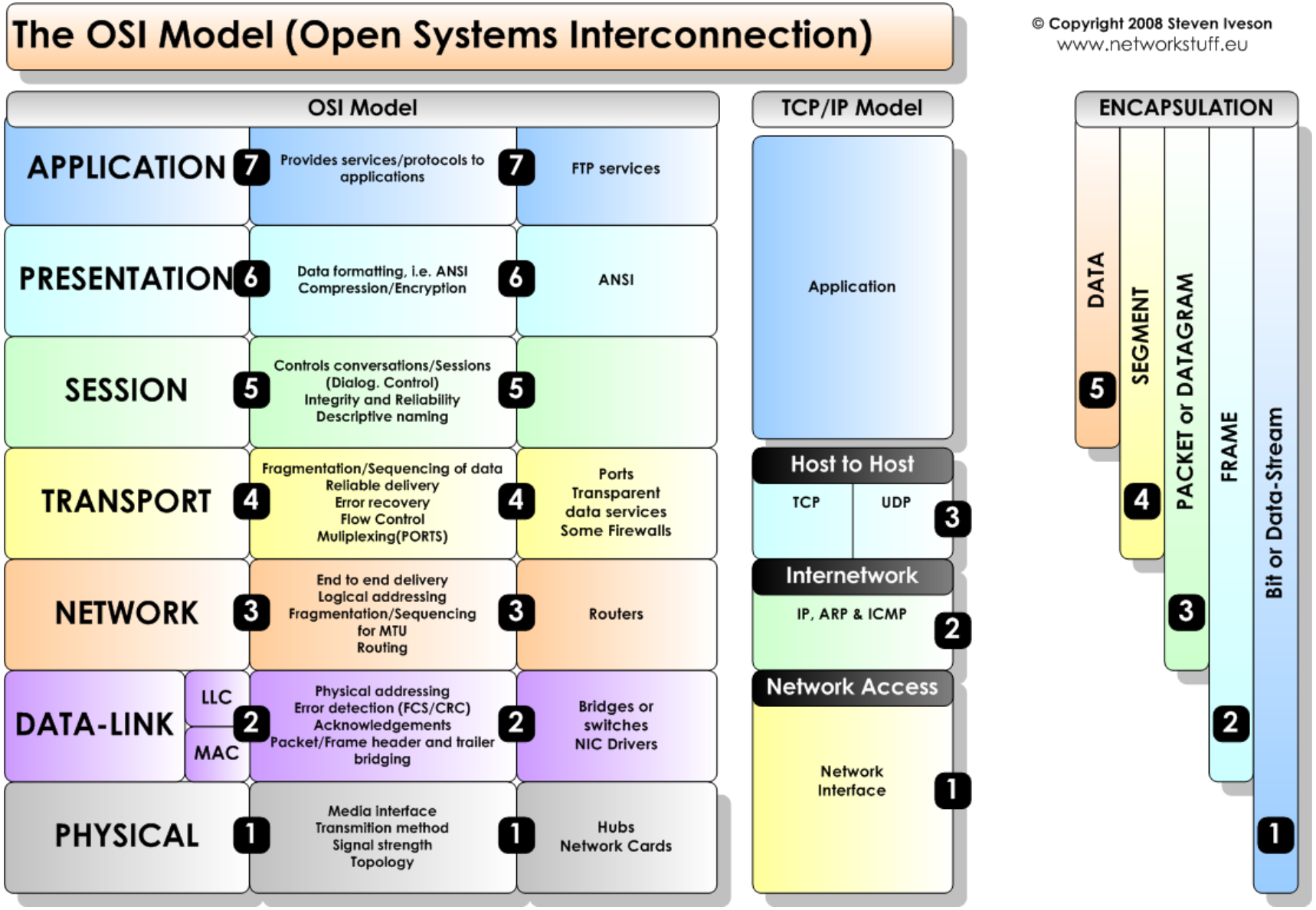
Registered Claims
Claims that are not mandatory whose names are reserved for us. These include:

- iss: The issuer of the token
- sub: The subject of the token
- aud: The audience of the token
- exp: This will probably be the registered claim most often used. This will define the expiration in Numeric Date value. The expiration MUST be after the current date/time.
- nbf: Defines the time before which the JWT MUST NOT be accepted for processing
- iat: The time the JWT was issued. Can be used to determine the age of the JWT
- jti: Unique identifier for the JWT. Can be used to prevent the JWT from being replayed. This is helpful for a one time use token.

Public Claims
These are the claims that we create ourselves like user name, information, and other important information.

Private Claims
A producer and consumer may agree to use claim names that are private. These are subject to collision, so use them with caution.

Source: https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication


The OSI Model (Open Systems Interconnection)
© Copyright 2008 Steven Iveson
www.networkstuff.eu
OSI Model
APPLICATION 7
Provides services/protocols to applications
7
FTP services
PRESENTATION 6
Data formatting, i.e. ANSI Compression/Encryption
6
ANSI
SESSION 5
Controls conversations/Sessions (Dialog. Control) Integrity and Reliability Descriptive naming
5
TRANSPORT 4
Fragmentation/Sequencing of data Reliable delivery Error recovery Flow Control Muliplexing(PORTS)
4
Ports Transparent data services Some Firewalls
NETWORK 3
End to end delivery Logical addressing Fragmentation/Sequencing for MTU Routing
3
Routers
DATA-LINK
LLC
MAC
2
Physical addressing Error detection (FCS/CRC) Acknowledgements Packet/Frame header and trailer bridging
2
Bridges or switches NIC Drivers
PHYSICAL 1
Media Interface Transmition method Signal strength Topology
1
Hubs Network Cards
TCP/IP Model
Application
Host to Host
TCP
UDP
3
Internetwork
IP, ARP & ICMP
2
Network Access
Network Interface
1
ENCAPSULATION
DATA
5
SEGMENT
4
PACKET or DATAGRAM
3
FRAME
2
Bit or Data-Stream
1